

Universidade Federal de Campina Grande  
Centro de Engenharia Elétrica e Informática  
Coordenação de Pós-Graduação em Ciência da Computação

## Scaling Testing of Refactoring Engines

Melina Mongiovi Cunha Lima Sabino

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

©Melina Mongiovi Cunha Lima Sabino, 29/11/2016

**FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG**

S116s Sabino, Melina Mongiovi Cunha Lima.  
Scaling testing of refactoring engines / Melina Mongiovi Cunha  
Lima Sabino. – Campina Grande, 2016.  
178 f. : il. color.

Tese (Doutorado em Ciência da Computação) – Universidade  
Federal de Campina Grande, Centro de Engenharia Elétrica e  
Informática, 2016.  
"Orientação: Prof. Dr. Rohit Gheyi".  
Referências.

1. Engenharia de Software. 2. Refatoramentos. 3. Testes de  
Software. 4. Geração de Programas. I. Gheyi, Rohit. II. Título.

CDU 004.41(043)

## Resumo

Definir e implementar refatoramentos não é uma tarefa trivial, pois é difícil definir todas as pré-condições necessárias para garantir que a transformação preserve o comportamento observável do programa. Com isso, ferramentas de refatoramentos podem ter condições muito fracas, condições muito fortes e podem aplicar transformações que não seguem a definição do refatoramento. Na prática, desenvolvedores escrevem casos de testes para checar suas implementações de refatoramentos e se preocupam em evitar esses tipos de bugs, pois 84% das asserções de testes do Eclipse e JRRT testam as ferramentas com relação aos bugs citados anteriormente. No entanto, as ferramentas ainda possuem esses bugs. Existem algumas técnicas automáticas para testar ferramentas de refatoramentos, mas elas podem ter limitações relacionadas com tipos de bugs que podem ser detectados, geração de entradas de testes, automação e performance. Este trabalho propõe uma técnica para escalar testes de ferramentas de refatoramentos. A técnica contém DOLLY um gerador automático de programas Java e C, no qual foram adicionadas mais construções de Java (classes e métodos abstratos e interface) e uma estratégia de pular algumas entradas de testes com o propósito de reduzir o tempo de testar as implementações de refatoramentos. Foi proposto um conjunto de oráculos para avaliar a corretude das transformações, dentre eles SAFEREFACTORIMPACT que identifica falhas relacionadas com mudanças comportamentais. SAFEREFACTORIMPACT gera testes apenas para os métodos impactados pela transformação. Além disso, foi proposto um novo oráculo para identificar transformações que não seguem a definição do refatoramento e uma nova técnica para identificar condições muito fortes. A técnica proposta foi avaliada em 28 implementações de refatoramentos de Java (Eclipse e JRRT) e C (Eclipse) e detectou 119 bugs relacionados com erros de compilação, mudanças comportamentais, condições muito fortes, e transformações que não seguem a definição do refatoramento. Usando pulos de 10 e 25 no gerador de programas, a técnica reduziu em 90% e 96% o tempo para testar as implementações de refatoramentos, enquanto deixou de detectar apenas 3% e 6% dos bugs, respectivamente. Além disso, detectou a primeira falha geralmente em alguns segundos. Por fim, com o objetivo de avaliar a técnica proposta com outras entradas de testes, foram avaliadas implementações do Eclipse e JRRT usando os programas de entrada das suas coleções de testes. Neste estudo, nossa técnica detectou mais 31 bugs não detectados pelos desenvolvedores das ferramentas.

## Abstract

Defining and implementing refactorings is a nontrivial task since it is difficult to define preconditions to guarantee that the transformation preserves the program behavior. Therefore, refactoring engines may have overly weak preconditions, overly strong preconditions, and transformation issues related to the refactoring definition. In practice, developers manually write test cases to check their refactoring implementations. We find that 84% of the test suites of Eclipse and JRRT are concerned with identifying these kinds of bugs. However, bugs are still present. Researchers have proposed a number of techniques for testing refactoring engines. Nevertheless, they may have limitations related to the bug type, program generation, time consumption, and number of refactoring engines necessary to evaluate the implementations. In this work, we propose a technique to scale testing of refactoring engines by extending a previous technique. It automatically generates programs as test inputs using Dolly, a Java and C program generator. We add more Java constructs in DOLLY, such as abstract classes and methods and interface, and a skip parameter to reduce the time to test the refactoring implementations by skipping some consecutive test inputs. Our technique uses SAFEREFACTORIMPACT to identify failures related to behavioral changes. It generates test cases only for the methods impacted by a transformation. Also, we propose a new oracle to evaluate whether refactoring preconditions are overly strong by disabling a subset of them. Finally, we present a technique to identify transformation issues related to the refactoring definition. We evaluate our technique in 28 refactoring implementations of Java (Eclipse and JRRT) and C (Eclipse) and find 119 bugs related to compilation errors, behavioral changes, overly strong preconditions, and transformation issues. The technique reduces the time in 90% and 96% using skips of 10 and 25 in Dolly while missing only 3% and 6% of the bugs, respectively. Additionally, it finds the first failure in general in a few seconds using skips. Finally, we evaluate our proposed technique by using other test inputs, such as the input programs of Eclipse and JRRT refactoring test suites. We find 31 bugs not detected by the developers.

## Agradecimentos

Gostaria de agradecer primeiramente a Deus, pelo dom da vida e pela benção da saúde;

Ao meu pai, Giuseppe Mongiovi, o grande incentivador da minha história, por todo amor, dedicação e companheirismo. Mesmo em memória, pude sentir sua constante presença, me guiando e me ajudando como sempre fez toda sua vida;

À minha mãe, Graça Mongiovi, pelo grande carinho, amor, preocupação e apoio, não medindo esforços para que eu conquistasse mais esta etapa da minha vida. Obrigada por estar sempre presente, vibrando com minha felicidade e me dando forças nos momentos difíceis;

Agradeço Gusthavo e Camilla, pelo amor, compreensão das minhas ausências e apoio em todos os momentos. Aos meus irmãos Finuzza e Angelo por sempre me encorajarem e torcerem por mim;

Agradeço ao meu amigo e orientador, Rohit Gheyi, pelos valiosos ensinamentos, paciência e dedicação durante meu doutorado. É admirável sua vontade de transmitir seu conhecimento e proporcionar o crescimento técnico do aluno como pesquisador e profissional. Sem ele esse trabalho não seria possível. Muito obrigada por tudo, Rohit;

Ao meu amigo Gustavo Soares, que acompanhou toda minha caminhada para realização deste trabalho. Obrigada pelo constante incentivo e ajuda durante o doutorado;

Aos professores Márcio Ribeiro, Leopoldo Teixeira, Paulo Borba e Tiago Massoni pelas valiosas sugestões e contribuições neste trabalho;

Agradeço a todos os amigos do SPG, em especial a Larissa Braz, Reudismam Rolim, Flávio Medeiros e Felipe Pontes, que contribuíram de alguma forma para melhorar essa pesquisa com boas conversas e trocas de ideias;

Aos professores e funcionários da COPIN e do DSC;

À Capes pelo apoio e suporte financeiro fornecidos a este trabalho.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem . . . . .	2
1.2	Solution . . . . .	4
1.3	Evaluation . . . . .	6
1.4	Summary of Contributions . . . . .	8
1.5	Organization . . . . .	9
<b>2</b>	<b>Background</b>	<b>10</b>
2.1	Program Refactoring . . . . .	10
2.1.1	Example . . . . .	11
2.1.2	Conditions . . . . .	11
2.1.3	Equivalence Notion . . . . .	12
2.2	Testing Overview . . . . .	14
2.2.1	Definition . . . . .	14
2.2.2	Test Data Adequacy . . . . .	15
2.2.3	Test Case Generation . . . . .	16
2.2.4	Testing Refactoring Engines . . . . .	18
2.3	Alloy Overview . . . . .	23
2.4	Technique to Test Refactoring Engines . . . . .	25
2.4.1	Overview . . . . .	25
2.4.2	Test Input Generation . . . . .	25
2.4.3	Refactoring Application . . . . .	28
2.4.4	Test Oracle . . . . .	28
2.4.5	Bug Categorizer . . . . .	29

---

<b>3</b>	<b>SAFEREFACTORIMPACT</b>	<b>32</b>
3.1	Motivating Example . . . . .	32
3.2	SafeRefactorImpact . . . . .	34
3.2.1	Change Impact Analysis . . . . .	35
3.2.2	Test Generation . . . . .	38
3.2.3	Change Coverage and Relevant Tests . . . . .	38
3.3	Evaluation . . . . .	40
3.3.1	Definition . . . . .	40
3.3.2	Planning . . . . .	41
3.3.3	Results . . . . .	43
3.3.4	Discussion . . . . .	46
3.3.5	Threats to Validity . . . . .	52
3.3.6	Answers to the Research Questions . . . . .	55
<b>4</b>	<b>A Technique to Scale Testing of Refactoring Engines</b>	<b>58</b>
4.1	Overview . . . . .	58
4.2	DOLLY 2.0 . . . . .	59
4.2.1	New Java Program Constructs . . . . .	60
4.2.2	Skipping Programs . . . . .	62
4.3	Evaluation . . . . .	63
4.3.1	Definition . . . . .	64
4.3.2	Planning . . . . .	64
4.3.3	Results . . . . .	66
4.3.4	Discussion . . . . .	68
4.3.5	Threats to Validity . . . . .	72
4.3.6	Answers to the Research Questions . . . . .	72
<b>5</b>	<b>Detecting Transformation Issues in Refactoring Engines</b>	<b>74</b>
5.1	Motivating Examples . . . . .	74
5.2	Oracles . . . . .	77
5.2.1	Differential Testing Oracle . . . . .	77
5.2.2	Structural Change Analysis Oracle . . . . .	78

---

5.3	Issue Categorization . . . . .	81
5.3.1	Technique using DT oracle . . . . .	81
5.3.2	Technique using the SCA oracle . . . . .	82
5.4	Evaluation . . . . .	82
5.4.1	Definition . . . . .	82
5.4.2	Planning . . . . .	83
5.4.3	Results . . . . .	83
5.4.4	Discussion . . . . .	84
5.4.5	Threats to Validity . . . . .	88
5.4.6	Answers to the Research Questions . . . . .	89
<b>6</b>	<b>Detecting Overly Strong Preconditions in Refactoring Engines</b>	<b>91</b>
6.1	Motivating Example . . . . .	91
6.2	Detecting Overly Strong Preconditions by Disabling Preconditions . . . . .	93
6.2.1	Overview . . . . .	94
6.2.2	Example . . . . .	95
6.2.3	Identifying Messages . . . . .	99
6.2.4	Disabling Refactoring Preconditions . . . . .	100
6.2.5	Evaluating Preconditions . . . . .	104
6.3	Evaluation . . . . .	105
6.3.1	Research Questions . . . . .	105
6.3.2	Planning . . . . .	106
6.3.3	Summary of the Results . . . . .	107
6.3.4	Discussion . . . . .	108
6.3.5	Threats to Validity . . . . .	123
6.3.6	Answers to the Research Questions . . . . .	125
<b>7</b>	<b>Related Work</b>	<b>127</b>
7.1	Refactoring . . . . .	127
7.2	Automated Testing of Refactoring Engines . . . . .	132
7.3	Change Impact Analysis . . . . .	135



---

<b>8</b>	<b>Conclusions</b>	<b>137</b>
8.1	Future Work . . . . .	141
<b>A</b>	<b>Survey – Implementing and Testing Refactorings</b>	<b>154</b>

# List of Figures

2.1	Technique for testing refactoring engines. . . . .	25
2.2	UML class diagram of JDOLLY’s meta-model. . . . .	27
3.1	Applying the Rename Intertype Declaration refactoring of Eclipse 4.2 with AJDT 2.2.3 leads to a behavioral change. . . . .	33
3.2	SAFEREFACTORIMPACT’s technique. . . . .	35
3.3	Distribution of the total time to evaluate transformations by SAFEREFACTOR and SAFEREFACTORIMPACT. . . . .	56
3.4	Distribution of the change coverage of the tests generated by SAFEREFACTOR and SAFEREFACTORIMPACT. . . . .	56
3.5	Distribution of the percentage of relevant tests generated by SAFEREFACTOR and SAFEREFACTORIMPACT. . . . .	57
4.1	A technique for scaling testing of refactoring engines. . . . .	59
4.2	Technique used by DOLLY 2.0 to skip programs. . . . .	63
5.1	Pulling up method B.m() using Eclipse JDT 4.5 does not remove the method from its original class. . . . .	75
5.2	Pushing down method A.m() using JRRT (02/03/13) removes a class from the program. . . . .	76
5.3	Differential Testing oracle. In Step 1 the oracle executes SAFEREFACTORIMPACT in both pairs of programs related to transformations of same refactoring type. If the output programs compile and preserve the program behavior, it compares the outputs concerning their AST (Step 2). The oracle reports the differences between the outputs if they exist. . . . .	78

---

5.4	Structural Change Analysis oracle. In Step 1 the oracle executes SAFEREFACTORIMPACT in the pair of programs related to a refactoring transformation. If the output program compiles and preserves the program behavior, the oracle checks the transformation concerning its refactoring definition. . . . .	79
5.5	Pulling Up Field B.f using Eclipse JDT 4.5 removes field C.f. . . . .	85
5.6	Encapsulating field B.f using Eclipse JDT 4.5 does not implement a correct <i>get</i> method because there is a method with the same signature. . . . .	87
6.1	A technique to identify overly strong preconditions. First, we generate the programs using JDOLLY (Step 1). For each generated program, we try to apply the transformation (Step 2). Next, for each kind of message reported by the engine, we inspect its code and manually identify the refactoring preconditions that can raise it. We perform transformations in the refactoring engine code to allow disabling the execution of the identified preconditions that prevent the refactoring application (Step 3). Then, for each rejected transformation we try to apply the transformation again using the refactoring engine with the preconditions that raise the reported messages disabled (Step 4). If the transformation preserves the program behavior according to SAFEREFACTORIMPACT, we classify the disabled preconditions as overly strong (Step 5). . . . .	94
7.1	Pulling up a field introduces a behavioral change in Eclipse JDT 4.3. . . . .	133

# List of Tables

1.1	Comparison between techniques to test refactoring engines. Express. = Expressiveness of the test inputs; CE = Compilation Errors; BC = Behavioral Changes; OSC = Overly Strong Conditions; TI = Transformation Issues; Yes = the technique contains the oracle/program generator; No = otherwise; CIA = Change Impact Analysis. . . . .	7
2.1	Filters for classifying behavioral changes. . . . .	30
3.1	Small-grained transformations considered by SAFIRA. . . . .	37
3.2	Results using a time limit of 0.2s. Methods = number of methods passed to Randoop to generate tests; Time = the total time of the analysis in seconds; Change Coverage = the percentage of impacted methods covered; Relevant Tests = the percentage of relevant tests; Result = it states whether the transformation is behavior-preserving. . . . .	44
3.3	Results using a time limit of 0.5s. Impacted Methods = number of methods identified by SAFIRA; Methods = number of methods passed to Randoop to generate tests; Time = the total time of the analysis in seconds; Change Coverage = the percentage of impacted methods covered; Relevant Tests = the percentage of relevant tests; Result = it states whether the transformation is behavior preserving. . . . .	45

3.4	Results using a time limit of 0.2s. Impacted Methods = number of methods identified by SAFIRA; Methods = number of methods passed to Randoop to generate tests; Time = the total time of the analysis in seconds; Change Coverage = the percentage of impacted methods covered; Relevant Tests = the percentage of relevant tests; Result = it states whether the transformation is behavior preserving. . . . .	45
3.5	Results using a time limit of 20s. Impacted Methods = number of methods identified by SAFIRA; Methods = number of methods passed to Randoop to generate tests; Time = the total time of the analysis in seconds; Change Coverage = the percentage of impacted methods covered; Relevant Tests = the percentage of relevant tests; Result = it states whether the transformation is behavior preserving. . . . .	46
3.6	Statistical analysis for defective refactoring data. Shapiro Test = analyze data normality; T-test = evaluate hypothesis test when data are normal; Wilcoxon-test = evaluate hypothesis test when data are non-normal; Result = final results of the statistical analysis; SR = SAFEREFACTOR; SRI = SAFEREFACTORIMPACT. . . . .	49
3.7	Statistical analysis for design patterns data. Shapiro Test = analyze data normality; Wilcoxon-test = evaluate hypothesis test when data are non-normal; Result = final results of the statistical analysis; SR = SAFEREFACTOR; SRI = SAFEREFACTORIMPACT. . . . .	50
3.8	Statistical analysis for larger subjects data. Shapiro Test = analyze data normality; T-test = evaluate hypothesis test when data are normal; Wilcoxon-test = evaluate hypothesis test when data are non-normal; Result = final result of the statistical analysis; SR = SAFEREFACTOR; SRI = SAFEREFACTORIMPACT. . . . .	53
4.1	Summary of the number of generated programs and the time to evaluate the refactoring implementations. . . . .	66
4.2	Summary of the detected bugs related to compilation errors. . . . .	66
4.3	Summary of the detected bugs related to behavioral changes. . . . .	67

4.4	Summary of the detected bugs related to overly strong preconditions using DT technique. . . . .	67
4.5	Summary of the Time to Find the First Failure (TTFF). . . . .	68
5.1	Changes considered by our program comparator. . . . .	79
5.2	Summary of the evaluation results of Eclipse and JRRT refactoring implementations with our technique using the SCA oracle; Refactoring = kind of refactoring; Scope = scope used by DOLLY to generate programs; P = package; C = class; M = method; F = field; Alloy Instances = number of Alloy instances generated by the Alloy Analyzer; GP (using skip of 25) = number of generated programs using skip of 25 in DOLLY 2.0; CP = compilable generated programs; Transformation Issues = number of different kinds of issues related to incorrect transformations; Time = total time to evaluate the refactoring implementations. . . . .	84
5.3	Summary of the evaluation results of Eclipse and JRRT refactoring implementations with our technique using DT oracle; GP (using skip of 25) = number of generated programs using skip of 25 in DOLLY 2.0; App. = applied transformations; BP = behavioral preserving applied transformations; Reject. by = rejected transformations; Different/Equal transf. applied = the outputs of the engines are different/equal; Transf. Issues = number of different kinds of issues related to incorrect transformations. . . . .	85
6.1	Summary of the DP technique evaluation in the JRRT and Eclipse refactoring implementations; Refact. = Kind of Refactoring; Skip = Skip value used by JDOLLY to reduce the number of generated programs; GP = Number of Generated Programs by JDOLLY; CP = rate of compilable programs (%); $N^o$ of assessed preconditions = Number of assessed refactoring preconditions in our study; Overly Strong Preconditions = Number of detected overly strong preconditions in the refactoring implementations; Time (h) = Total time to evaluate the refactoring implementations in hours; Time to First Failure (min) = Time to find the first failure in minutes; "na" = not assessed.	109

6.2	Summary of the comparison between DP and DT techniques using input programs generated by JDOLLY; Refact. = Kind of Refactoring; Skip = Skip value used by JDOLLY to reduce the number of generated programs; DP = DP Technique; DT = DT Technique; Overly Strong Preconditions = Number of detected overly strong preconditions in the refactoring implementations; "na" = not assessed. . . . .	110
6.3	Subset of Eclipse and JRRT assessed preconditions. Engine = Refactoring engine that contains the precondition; Refactoring = Kind of refactoring; Precondition = precondition checking; Message = reported message when the precondition is unsatisfied; OS (DP) = yes if the DP technique found this precondition as overly strong in this experiment, otherwise no. . . . .	112
6.4	Summary of the comparison between DP and DT techniques using input programs of Eclipse and JRRT refactoring test suite; Refactoring = Kind of Refactoring; Input programs = Number of selected input programs of the JRRT and Eclipse refactoring test suite; $N^o$ of assessed preconditions = Number of assessed refactoring preconditions in our study; Overly Strong Preconditions = Number of detected overly strong preconditions in the refactoring implementations; DP = DP Technique; DT = DT Technique. . . . .	122
6.5	Summary of the evaluation results of our technique to detect overly weak preconditions using input programs of Eclipse and JRRT refactoring test suite; Refactoring = Kind of Refactoring; Input programs = Number of selected input programs of the JRRT and Eclipse refactoring test suite; Compilation Errors = Number of detected bugs related to compilation errors in the refactoring implementations; Behavioral Changes = Number of detected bugs related to behavioral changes in the refactoring implementations. . . . .	123

# List of Source Codes

2.1	Program with bad smell: long method. . . . .	12
2.2	Method <i>printDebt()</i> of the refactored program 2.1. . . . .	13
2.3	Test case generated by Randoop to test the API of the JDK <i>Collection</i> . . . . .	17
2.4	Test cases of JRRT (02/03/13) to evaluate the Push Down Method refactoring. . . . .	19
2.5	Before refactoring. . . . .	21
2.6	After refactoring: applying Rename Field in Eclipse JDT 3.7 leads to a compilation error due to field hiding. . . . .	21
2.7	Before refactoring. . . . .	22
2.8	After refactoring: applying Pull Up Method in Eclipse JDT 3.7 leads to a behavioral change due to incorrect change of <b>super</b> to <b>this</b> . . . . .	22
2.9	Original version. . . . .	23
2.10	Correct target's version after removing a subset of overly strong conditions. . . . .	23
2.11	Original program. . . . .	29
2.12	Resulting program. . . . .	29
2.13	Renaming the <i>n</i> method to <i>k</i> is not allowed using Eclipse 3.7. . . . .	31
3.1	Original program. . . . .	33
3.2	Resulting program. . . . .	33
3.3	Test suite of the program presented in Listing 3.1. . . . .	34
3.4	A unit test revealing a behavioral change in the transformation presented in Figure 3.1. . . . .	39
3.5	A non-relevant unit test generated by Randoop in SAFEREFACTOR used to evaluate the transformation presented in Figure 3.1. . . . .	39
3.6	Original program. . . . .	48
3.7	Resulting program after the transformation applied in Subject 2. . . . .	48



---

4.1	Uncompilable output program generated by the Push Down Field refactoring of Eclipse JDT 4.3. . . . .	69
4.2	Before refactoring. . . . .	70
4.3	After the Pull Up Field Refactoring of Eclipse JDT 4.3. . . . .	70
4.4	Before refactoring. . . . .	71
4.5	After the Move Method Refactoring of Eclipse JDT 4.3. . . . .	71
5.1	Original version. . . . .	75
5.2	Resulting program. . . . .	75
5.3	Original version. . . . .	76
5.4	Resulting program. . . . .	76
5.5	Resulting program after Eclipse JDT 4.5 pushes down method A.m() in the program presented in Listing 5.3. . . . .	76
5.6	Original version. . . . .	85
5.7	Resulting program. . . . .	85
5.8	Original version. . . . .	87
5.9	Resulting program. . . . .	87
6.1	Pulling Up doQuery method is rejected by Eclipse JDT 2.1. . . . .	93
6.2	Correct resulting program version. . . . .	93
6.3	Pulling Up method B.m() to class A is rejected by Eclipse JDT 4.6. . . . .	96
6.4	A possible correct resulting program version. . . . .	96
6.5	Original code snippet of Eclipse JDT. . . . .	98
6.6	Code snippet after disabling a Pull Up Method refactoring precondition using Transformation 2. . . . .	98
6.7	Abstract aspect to disable preconditions. . . . .	103
6.8	Aspect to disable refactoring preconditions in Eclipse. . . . .	105
6.9	Original code snippet of JRRT. . . . .	114
6.10	Code snippet after we change the engine code by using Transformation 1, to allow disabling the execution of a Move Method refactoring precondition. . . . .	114
6.11	Pushing down field A.f to class C is rejected by JRRT. Bug detected by DP technique and not detected by DT technique because Eclipse also rejects to apply the transformation. . . . .	119

---

6.12	A possible correct resulting program version applied by JRRT. . . . .	119
6.13	Renaming C.f0 to f1 is rejected by Eclipse JDT 4.5. Bug detected by DT technique and not detected by DP technique. . . . .	120
6.14	A possible correct resulting program version applied by JRRT. . . . .	120
	aName . . . . .	125
7.1	Original Program . . . . .	133
7.2	Resulting Program . . . . .	133

# Chapter 1

## Introduction

During its life cycle, a software may change due to introduce new features and enhancements, improve its internal structure, or make its processing more efficient. Systems continue to evolve over time and become more complex as they grow. Developers can take some actions to avoid that, such as code refactoring, a kind of perfective maintenance [82]. The term Refactoring was originally coined by Opdyke and Johnson [56], and popularized in practice by Fowler [14], as the process of changing the internal structure of a program to improve its internal quality while preserving its external behavior.

Refactorings can be manually applied, which may be time consuming and error prone, or automatically by using a refactoring engine, such as Eclipse [12], NetBeans [54], and JstAdd Refactoring Tools (JRRT) [68]. Refactoring engines may contain a number of refactoring implementations, such as Rename Class, Pull Up Method, and Encapsulate Field. For correctly applying a refactoring, and thus ensure behavior preservation, the refactoring implementations usually need to consider preconditions, such as checking for naming conflicts. However, defining and implementing refactorings is a nontrivial task since it is difficult to define all preconditions to guarantee that the transformation<sup>1</sup> preserves the program behavior. In fact, proving refactoring correctness for entire languages, such as Java and C, constitutes a challenge [70].

In practice, refactoring engine developers may implement the refactoring preconditions based on their experience, some previous work [16] or formal specifications [68]. How-

---

<sup>1</sup>Hereafter, we refer to transformation as a modification to a program and refactoring transformation as a transformation that preserves the program behavior.

ever, the implemented preconditions may be overly weak, allowing non-behavior preserving transformations or overly strong, preventing developers from applying useful transformations. Also, the implementation may not follow the refactoring definition [64; 55; 14; 68] (what a specific kind of refactoring transformation must and must not do). Then, refactoring engines may have bugs [80; 77].

## 1.1 Problem

In general, developers of refactoring engines manually write test cases to detect overly weak preconditions, overly strong preconditions, and transformations that do not follow the refactoring definition (transformation issues), which may be time consuming and error prone. We investigate the test suites of 20 refactoring implementations of Eclipse JDT 4.5 and JRRT (02/03/13) and find that 84% of the test assertions are concerned with identifying those kinds of bugs. Nevertheless, the bugs are still present. Testing refactoring engines is not trivial since it requires complex inputs, such as programs, and an oracle to define the correct resulting program or whether the transformation must be rejected. Manually writing test cases may be costly, and thus it may be difficult to create a good test suite considering all language constructs. Researchers have proposed a number of automated techniques for testing refactoring engines [10; 22; 77; 21]. They may automate four major steps of the testing process: (i) generating test inputs; (ii) applying the refactoring; (iii) checking the output correctness; (iv) and classifying the detected failures into distinct bugs.

For example, to automate test input generation, Gligoric et al. [22] propose UDITA (an extension of ASTGEN [10]), a Java-like language to write program generators so that developers can generate programs as test inputs. They used UDITA to generate about 5,000 programs with up to 3 classes as test inputs [22]. However, as the authors of UDITA stated later [21], configuring UDITA to generate specific programs demands a considerable effort. Soares et al. [80; 77] propose a Java program generator called JDOLLY for exhaustively generating programs. By using JDOLLY, developers can specify the number of some Java constructs and constraints for the generated programs by using Alloy [27], a formal specification language. They used JDOLLY to generate more than 100,000 programs. Alloy logic presented, as expected, a higher level of abstraction than Java-like code. For exam-

ple, the results of the closure operator in Alloy can only be achieved programmatically after considerable additional effort. Their technique has an oracle that uses Differential Testing (DT) to automatically identify overly strong preconditions in refactoring implementations.<sup>2</sup> It applies the same refactoring to each test input using two different implementations, and compares the results. The DT technique needs at least two refactoring engines. When both engines under test reject to apply a transformation due to overly strong preconditions the DT technique cannot detect them. Additionally, automating another engine may be costly, as the developer might not be familiar with its code. Finally, this approach can only be used if the engines implement the same refactoring. Exhaustively generating programs, even for a small number of Java constructs, may require a lot of time. To alleviate this problem, Jagannath et al. [29] propose the Sparse Test Generation technique (STG), which skips some test inputs. They reduce the time to find the first failure.

Later, Gligoric et al. [21] propose to use real programs as test inputs, automatically applying the refactoring under test in every possible location of the program. They found 141 bugs related to compilation errors and engine crash in the refactoring implementations of Eclipse JDT and CDT by using 8 real systems in Java and C as test inputs. Although this approach can reduce the effort to create test inputs, testing refactoring engines in large programs may increase the costs of checking the output correctness. For example, to identify bugs related to behavioral changes, Soares et al. [77] use SAFEREFACTOR, a tool that analyzes a transformation and generates tests to compare the program behavior before and after the transformation. SAFEREFACTOR was useful for finding 63 bugs related to behavioral changes in the programs generated by JDOLLY. However, using SAFEREFACTOR to evaluate transformations on large real programs would require a much higher time for analyzing the transformation and generating tests. Additionally, understanding a failure in a large transformation demands more time. Gligoric et al. [21] take 1-60 minutes to analyze each failure in order to categorize them into distinct bugs. In summary, the previous approaches have limitations related to the kinds of bugs that can be detected, program generator (exhaustiveness, setup, expressiveness), time consumption, or number of refactoring engines necessary to evaluate a refactoring implementation.

---

<sup>2</sup>Hereafter, we refer to the technique that uses DT oracle as DT technique.

## 1.2 Solution

In this work, we propose a technique to scale testing of refactoring engines by improving limitations of techniques discussed in the previous section [77; 80; 22; 21]. It automatically generates programs as test inputs using DOLLY, an automated and exhaustive Java and C program generator. Our technique can find bugs related to overly weak preconditions (compilation errors and behavioral changes), overly strong preconditions, and transformation issues. We improve the previous technique [77] with respect to DOLLY's expressiveness, reduction of the time to test the refactoring implementations, and new oracles to detect behavioral changes and transformation issues. We add more Java constructs in DOLLY to improve its expressiveness, propose a technique to skip some consecutive test inputs to reduce the costs and improve performance [47], present a new technique to identify overly strong preconditions that does not need other refactoring engine, refine an oracle to identify behavioral changes [46], and introduce two oracles to identify a new kind of bug related to transformation issues.

Our technique may reduce the time to test the refactoring implementations by skipping some consecutive test inputs. Consecutive programs generated by DOLLY tend to be very similar, potentially detecting the same kind of bug. Thus, developers can set a parameter to skip some programs to reduce the time to test the refactoring implementations. By skipping these programs, we can reduce the Time to First Failure (TTFF), reducing the developer idle time [29]. We improve the expressiveness of DOLLY by adding abstract classes, abstract methods, and interfaces. By improving the expressiveness of the program generator, the technique may find more bugs. For example, all transformation issues that our technique finds in this work, were related to these new DOLLY's constructs.

We propose a new technique to identify overly strong preconditions by disabling some preconditions (DP technique). For each program generated by DOLLY, we apply the transformation using the refactoring engine under test. Next, we collect the different kinds of messages reported by the refactoring engine when it rejects transformations. For each kind of message, we inspect the refactoring engine and manually identify the refactoring preconditions that can raise it. We change the refactoring engine code to allow disabling the preconditions that prevent the refactoring. If the engine, with some preconditions disabled applies

the transformation, and it preserves the program behavior according to SAFEREFACTORIMPACT [46], then we classify the set of disabled preconditions as overly strong. SAFEREFACTORIMPACT [46] automatically checks whether a transformation preserves the program behavior.

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

We use SAFEREFACTORIMPACT as the oracle to detect behavioral change transformations. SAFEREFACTORIMPACT generates test cases only for the methods impacted by a transformation. It reduces the time to test the refactoring implementations and generates more relevant tests than SAFEREFACTOR. Previously [45], we proposed SAFEREFACTORIMPACT and used it to detect faults related to overly weak preconditions. In this work, we made minor improvements and also used it to detect faults related to overly strong preconditions and transformation issues. Also, we evaluated it in new subjects, including real case studies, considering Object-Oriented (OO) and Aspect-Oriented (AO) constructs with respect to two new defined metrics (change coverage and relevant tests), time to evaluate a transformation, and detected behavioral change transformations. The evaluation in the context of Aspects showed evidence that the technique is useful to evaluate transformations in AspectJ programs.

After identifying the failures, the proposed technique uses a set of automated bug categorizers to classify all failing transformations into distinct bugs. In our previous work [80] we used an approach similar to the approach proposed by Jagannath et al. [29] (Oracle-based Test Clustering) to automate the classification of failures related to overly strong precondi-

tions. We implement an automated issue categorizer to classify the outputs of DT and SCA oracles into different kinds of issues. It is based on the kinds of differences between the outputs (for DT oracle) and the kinds of refactoring definitions that the transformations do not follow (for SCA oracle). Soares et al. [77] specified a systematic, but manual approach to categorize failures related to behavioral changes. We automate it in this work. For simplicity we use the term transformation to refer to a refactoring or a failing transformation.

Table 1.1 summarizes the comparison among our technique and previous techniques to test refactoring engines. It illustrates an overview of previous technique's limitation and our contribution to improve some of them in this work. Our technique has DOLLY, an automated program generator for Java (JDOLLY [77]) and C (CDOLLY) (Column 2). DOLLY extends JDOLLY by adding more Java constructs to improve its expressiveness (Column 3). We also included a skip parameter to reduce the number of generated programs and thus, reduce the time to test the refactoring implementations (Column 4). Our technique uses SAFER-EFACTORIMPACT to automatically detect behavioral changes using change impact analysis (Column 6). We propose the DP technique to detect overly strong preconditions by disabling some of them. It does not need other refactoring engine that implements the same refactoring to test a refactoring implementation (Column 7). Finally, we propose a new automated oracle to detect transformation issues related to the refactoring definition (Column 8).

## 1.3 Evaluation

We evaluated our proposed technique to scale testing of refactoring engines in 28 refactoring implementations of JRRT [68], Eclipse JDT (Java), and Eclipse CDT (C). We found 119 bugs in a total of 49 bugs related to compilation errors, 17 bugs related to behavioral changes, 35 bugs related to overly strong preconditions (30 bugs using DP technique and 24 using DT technique), and 18 transformation issues related to the refactoring definition. We also compared the impact of the skip on the time consumption and bug detection in our technique. The technique reduces the time in 90% and 96% using skips of 10 and 25 in DOLLY while missing only 3% and 6% of the bugs, respectively. By using skips, we found the first failure related to compilation error, behavioral change, or overly strong preconditions in general in a few seconds. So, the refactoring engine developer can quickly find a bug in the refactoring



Table 1.1: Comparison between techniques to test refactoring engines. Express. = Expressiveness of the test inputs; CE = Compilation Errors; BC = Behavioral Changes; OSC = Overly Strong Conditions; TI = Transformation Issues; Yes = the technique contains the oracle/program generator; No = otherwise; CIA = Change Impact Analysis.

Technique	Test Inputs		Cost	Oracle			
	Automatic generation	Express.		CE	BC	OSC	TI
[10],[22],[29]	Yes	Medium	Medium	Yes	Yes/Syntactic	No	Yes/Custom oracle
[21]	Yes	High	High	Yes	No	No	No
[77],[80]	No	Low	High	Yes	Yes/Dynamic	Yes/two engines	No
Our technique	Yes	Medium	Medium	Yes	Yes/Dynamic w/ CIA	Yes/one engine	Yes/SCA

implementation, fix it, run our technique again to find another bug, and so on. Before a release, tool developers can run the technique without skip to find the missed bugs.

We evaluated the DP technique in 10 refactorings from Eclipse JDT 4.5 and the same 10 of the latest JRRT version (02/03/13). We generated more than 150,000 programs as test inputs and detect 30 overly strong preconditions in the refactoring implementations. So far, Eclipse developers confirmed 47% of them. It took around 1h and 35h to detect all overly strong preconditions of JRRT and Eclipse, respectively. Our current setup to test the refactoring implementations of Eclipse is costlier than the JRRT ones. The DP technique took on average a few seconds to find the first overly strong precondition in JRRT and on average 17.41 minutes in Eclipse.

We compared the DP technique with our previous one (DT technique) by using the same input programs. The DP technique detected 11 bugs (37% of new bugs) not detected by the DT technique, while missing 5 bugs (21% of the bugs detected by the DT technique). In addition, the DP technique did not require using another engine with the same refactorings to compare the results. So, whenever possible, developers can run the DP technique and after fixing the detected bugs, they run the DT technique to find more bugs.

We also performed another study in which we used programs from the Eclipse and JRRT

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

We evaluated our oracles to identify transformation issues in eight refactoring implementations of Eclipse JDT 4.5 and JRRT using DOLLY with abstract classes and methods, and interface. We scale the new version of DOLLY to deal with a million Alloy instances. We used skip of 25 to reduce the costs and found 10 transformation issues in Eclipse and 8 in JRRT.

## 1.4 Summary of Contributions

The main contributions of this work [46; 47] are the following:

- A technique to scale testing of refactoring engines by reducing the costs and improving bug detection:
  - New features in the program generator, DOLLY (Chapter 4);
    - \* New Java constructs, such as abstract classes and methods, and interface;
    - \* A skip mechanism to reduce the set of test inputs [47];
  - A new technique to identify overly strong preconditions in refactoring implementations by disabling some preconditions (Chapter 6);
  - Two oracles to identify transformation issues (Chapter 5);
  - An oracle to detect behavioral change transformations based on change impact analysis and test generation [46] (Chapter 3);

- An evaluation of the technique on testing 28 kinds of refactorings implemented by Eclipse JDT, Eclipse CDT, and JRRT with respect to overly weak (compilation errors and behavioral changes) and strong preconditions (using DT and DP techniques) using no skip and skips of 10 and 25 to generate programs (Chapters 4 and 6);
- An evaluation of our oracle to identify transformation issues in eight refactoring implementations of Eclipse JDT and JRRT (Chapter 5);
- A more extensive evaluation of SAFEREFACTORIMPACT in new subjects, including real case studies, considering OO and AO constructs with respect to two new defined metrics (change coverage and relevant tests), time and detected behavioral changes (Chapter 3).

## 1.5 Organization

This thesis is organized as follows. In Chapter 2, we provide some background on program refactoring, testing, Alloy, and a previous automated technique for testing refactoring engines [77; 80], which we use and extend in this work. In Chapter 3, we give an overview of SAFEREFACTORIMPACT, and present an evaluation of SAFEREFACTORIMPACT on a number of Java and AspectJ transformations and comparing it with SAFEREFACTOR. Chapter 4 presents our technique to scale testing of refactoring engines and an evaluation in 28 kinds of refactoring implementations with respect to compilation errors, behavioral changes, and overly strong preconditions using DT technique. In Chapter 6, we describe our new technique to identify overly strong preconditions in refactoring engines by disabling some preconditions. Moreover, we show its evaluation by testing real Java refactoring engines. Next, we explain our technique to identify transformation issues in Chapter 5. Chapter 7 presents related work. Finally, Chapter 8 summarizes the contributions of this thesis and presents future work.

# Chapter 2

## Background

In this chapter we present the background of some concepts needed for understanding this work. First, we explain program refactoring in Section 2.1. Next, Section 2.2 presents an overview about testing activities. Finally, we describe an Alloy overview in Section 2.3.

### 2.1 Program Refactoring

Opdyke originally coined the term refactoring in his PhD thesis [55]. Later, Fowler [14] popularized it. They define code refactoring as the process of modifying a software system in order to improve its internal quality while preserving the observable behavior. The essence of code refactoring consists of a number of small changes that preserve the program behavior. A sequence of small changes (known as refactorings) produces a substantial restructuring. According to Mens and Tourwé [44], the process of code refactoring consists of the following activities:

1. Identifying where the software should be refactored;
2. Determine which refactoring(s) should be applied to the identified places;
3. Guarantee that the applied refactoring preserves behavior;
4. Apply the refactoring;
5. Assess the effect of the refactoring on quality characteristics of the software or the process;

6. Maintain the consistency between the refactored program code and other software artifacts.

### 2.1.1 Example

Fowler [14] introduced the term *bad smells* to identify code structures that suggest the possibility of refactoring. Bad smells indicate that some code structures are not good and need to be improved. After identifying a bad smell, we need to apply an adequate refactoring in order to improve the code. Examples of bad smells are: duplicate code, long method, large class, long parameter list, and switch statements.

The following example presents a code with bad smell and an application of a refactoring to remove it. Listing 2.1 illustrates the original program with a bad smell. The program contains the *printDebt()* method that prints client debt. First, it prints the header that contains information about the client, such as name and address. Next, it computes the client debt by adding the cost of each order. Finally, the method prints the client orders and its total debt. The method contains a bad smell: it deals with many concerns and thus, needs a number of comments to facilitate its understanding. The Extract Method refactoring is appropriate to be applied. This kind of refactoring extracts parts of the method and makes a new method.

Listing 2.2 shows the refactored code. We extracted three methods: *printHeader*, *computeDebt*, and *printDetails*. After the refactoring, the method *printDebt()* became easier to understand and improved the chances of code reuse.

### 2.1.2 Conditions

Refactoring implementations must implement a set of preconditions to secure that the transformations preserve the program behavior. For example, the Rename Type refactoring changes the original name of a class to a new name proposed by the user. This refactoring implementation must check if there is another type in the same package with the new name. Otherwise, the transformation may generate a program that does not compile.

Listing 2.1: Program with bad smell: long method.

```
void printDebt () {
    List<Order> orders = client.getOrder ();
    double debt = 0.0;
    // print header
    System.out.println ("*** Debts ***");
    System.out.println ("Name: " + client.name);
    System.out.println ("Address: " + client.address);
    // compute debt
    for (Order order: orders) {
        debt += order.value ();
    }
    // print details
    System.out.println ("Order");
    printOrders (orders);
    System.out.println ("Debt value: " + debt);
}
```

However, defining all of the necessary preconditions to guarantee that a transformation preserves the program behavior or to prevent applying a behavioral change transformation is not trivial. Indeed, proving the correctness of the preconditions with respect to a formal semantics for complex languages such as Java and C, constitutes a challenge [70]. Refactoring engine developers use informal sets of preconditions as the basis for implementing refactorings. Then, the engines can have overly weak or overly strong conditions. Overly weak conditions allow applying transformations that change the program behavior, while overly strong conditions reject behavior preserving transformation.

### 2.1.3 Equivalence Notion

The term equivalence notion refers whether two programs have the same behavior. There are different notions about equivalence of programs. The notion depends upon the context in which it is applied. William Opdyke [55] defines semantic 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).*”

Listing 2.2: Method *printDebt()* of the refactored program 2.1.

```
void printDebt () {
    printHeader ();
    double debt = computeDebt ();
    printDetails (debt);
}
void printHeader () {
    System.out.println ("*** Debts ***");
    System.out.println ("Name: " + client.name);
    System.out.println ("Address: " + client.address);
}
double computeDebt () {
    List<Order> orders = client.getOrder ();
    double debt = 0.0;
    for (Order order: orders) {
        debt += order.value ();
    }
    return debt;
}
void printDetails (double debt) {
    System.out.println ("Order");
    printOrders (orders);
    System.out.println ("Debt value: " + debt);
}
```

Fowler's [14] notion is similar as the Opdyke's one: *"The second thing I want to highlight is that refactoring does not change the observable behavior of the software. The software still carries out the same function that it did before. Any user, whether an end user or another programmer, cannot tell that things have changed (p. 47)."*

However, as Don Roberts [64] writes, there are several problems with this informal definition because for some application domains the equivalence notion is not only related to methods outputs. For example, a program that changes the execution time of a routine by 10 milliseconds, without altering its inputs or outputs, would normally be considered behavior preserving. However, in a hard real-time application, such transformation is not considered behavior preserving. Another problem is related to embedded systems. In those systems,

properties like memory, space, and energy consumption may be crucial to define behavior preservation.

In practice, for most programs, developers use regression tests to evaluate if a program preserves its behavior after a change made in it. If the regression tests pass before and after a transformation, they consider that the program behavior was preserved. Based on this notion, Soares et al. [77] consider that a transformation preserves the program behavior when public methods with unchanged signatures (before and after a transformation) have the same outputs for the same inputs after a change in the program. They argue that 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 part of the overall behavior of the system under test; changes in these methods will not affect the system behavior. In this work, we use the same equivalence notion.

## 2.2 Testing Overview

In this section, we give an overview about software testing activities. Section 2.2.1 describes some concepts and definitions about testing. Next, Section 2.2.2 explains about test data adequacy. Section 2.2.3 presents some techniques of test case generation. Finally, we explain about testing of refactoring engines in Section 2.2.4.

### 2.2.1 Definition

A software system must be predictable and consistent, offering no surprise to the user. Software testing consists of a process of executing a program in order to assess whether it works correctly according to its specification [52]. The main objective of a test activity is to reveal many failures as possible with a minimal effort. Binder [3] clarifies some important concepts to software testing activity: *failure*, *fault*, and *system error*. A failure is an external, incorrect behavior with respect to the expected behavior; a fault is a static defect in the software; and a system error is an incorrect internal state (the manifestation of some fault).

The test execution takes place at different levels throughout the software development. According to Rock et al. [66], the main software levels are:

- Unit Test: aims to explore the smallest unit of the project. The goal is to search for



failures, in each module separately, caused by defects in the logic implementation. In this type of test, the target universe are the methods of objects or even small code snippets.

- **System Test:** aims to evaluate the entire software by using it as an end user. Thus, the tests are executed in the same environment, with the same conditions, and using the same inputs that a user may use in practice.
- **Integration Test:** aims to induce failures associated with the interfaces among module systems that were integrated to build the software structure established in the design phase.
- **Regression Test:** does not consist of a test level, but it is an important strategy to reduce the side effects of a change in the program. It is the process of testing changes to a program to make sure that it still works correctly with the new changes.

### 2.2.2 Test Data Adequacy

Testing is essential to control software quality. It is widely recognized in the software engineering the importance of tests to evaluate the system. But some questions about test activities remain: Is the test suite adequate? When to finish the test activity and integrate? Is all the possible scenarios being tested? A criteria of test data adequacy can help to answer those questions.

An important issue about managing a software testing activity is to ensure that we know and can define the goals of the tests in terms of what can be measured. A test criteria defines what constitutes an adequate test suite. It identifies the properties of a program needed to be exercised in order to ensure that the test suite is complete [23]. Most test criteria proposed in the literature explicitly specify requirements of the test. They are objective rules applied by project managers for this purpose [96]. For example, branch coverage is a testing requirement, which states that all branches of the program need to be exercised by the test suite. So if this is the test criteria used, the test goal should satisfy this requirement.

The code coverage measures the rate of code exercised by the test suite [23]. The main strategies to measure code coverage are *Statement Coverage*, *Branch Coverage*, and *Path*

*Coverage* [96]. *Statement Coverage* indicates the rate, from all program statements, of statements exercised by a test suite, whether it includes comments or not; *Branch Coverage* represents the rate of control transfers in the program executed by the test suite; and *Path Coverage* states the rate of execution paths from the program's entry to its exit executed by the test suite.

### 2.2.3 Test Case Generation

Manual testing activities are tedious, time consuming, and mainly, error prone. Manual tests may not be enough to make a quality testing activity. Unlike manual testing, automated tests can be performed whenever desired, requiring less human effort. Indeed, the main goal of test automation is to reduce the human effort in test activities [30]. Nowadays, there are some techniques to automatically generate test cases to a system program.

The main strategies to generate tests are: (i) systematic generation and (ii) random generation [24] of test cases. Within systematic generation, we highlight some techniques: exhaustive generation of test cases [42], chaining [13] and symbolic execution [90]. Other techniques have been proposed using evolutionary algorithms [2] and development guided by contracts [38]. In what follows, we describe Randoop [58] and EvoSuite [15], two automatic test generators.

#### **Randoop**

Randoop is an automatic unit test generator for Java. It automatically creates unit tests in JUnit format [58]. Randoop generates unit tests using feedback-directed random test generation. In a nutshell, this technique randomly, generates sequences of methods and constructor invocations for the classes under test, and uses the sequences to create tests. Randoop executes the sequences that it creates, using the results of the execution to create assertions that capture the behavior of the program and that catch bugs.

Randoop can generate two types of tests: error-revealing tests and regression tests. Error-revealing tests are tests that fail when executed, indicating a potential error in one or more classes under test. Regression tests are tests that pass when executed, and can be used to augment a regression test suite.

The error-revealing tests show the specific use of a tested class that violates a contract in execution time. A contract is a class or method property that must be preserved. The violation of a contract suggests code errors. For example, Listing 2.3 illustrates a test generated by Randoop revealing a violation of the contract *reflexivity of equality*. This test found a bug in the API of the JDK *Collection*. It shows that classes of *Collection* can create an object not equal to itself. A *TreeSet* is an ordered collection. According to the *Sun* API, a call to its constructor giving a *List* as parameter (line 5), must throw an exception, as it is not possible to compare the list elements. However, the constructor accepts the list as parameter. Therefore, this contract was violated and the assertion shown in line 7 fails in execution time.

Listing 2.3: Test case generated by Randoop to test the API of the JDK *Collection*.

```
1 public static void test() {
2     LinkedList list = new LinkedList();
3     Object o1 = new Object();
4     list.addFirst(o1);
5     TreeSet t1 = new TreeSet(list);
6     Set s1 = Collections.synchronizedSet(t1);
7     Assert.assertTrue(s1.equals(s1));
8 }
```

---

Randoop [58] considers the following contracts:

- *Equals to null*: *o.equals(null)* should return *false*;
- *Reflexivity of equality*: *o.equals(o)* should return *true*;
- *Symmetry of equality*: *o1.equals(o2)* implies *o2.equals(o1)*;
- *Equals-hashcode*: If *o1.equals(o2)==true*, then *o1.hashCode()==o2.hashCode()*;
- *No null pointer exceptions*: No *NullPointerException* is thrown if no *null* inputs are used in a test.

The regression tests generated by Randoop are useful to be run after a code change. If the test passes right after its generation and it fails after a code change, then the change may have altered the program behavior.

## EvoSuite

EvoSuite [15] is a tool that automatically generates test cases with assertions for classes written in Java. To achieve this, EvoSuite applies a hybrid approach that generates and optimizes whole test suites towards satisfying a coverage criterion. For the produced test suites, EvoSuite suggests possible oracles by adding small sets of assertions that concisely summarize the current behavior; these assertions allow the developer to detect deviations from expected behavior, and to capture the current behavior in order to protect against future defects breaking this behavior.

EvoSuite implements two novel techniques to achieve its objectives: complete test suite generation and mutation-based assertion generation. The whole test suite generation uses an evolutionary search approach that evolves whole test suites with respect to an entire coverage criterion at the same time. Optimizing with respect to a coverage criterion, rather than individual coverage goals, achieves that the result is neither adversely influenced by the order nor by the difficulty or infeasibility of individual coverage goals. The mutation-based assertion generation uses mutation testing to produce a reduced set of assertions that maximizes the number of seeded defects in a class that are revealed by the test cases. These assertions highlight the relevant aspects of the current behavior in order to support developers in identifying defects, and the assertions capture the current behavior to protect against regression faults.

### 2.2.4 Testing Refactoring Engines

In practice, developers of refactoring engines manually write test cases. To specify a test, the developer needs to create a program to be refactored, as test input. The developer also needs to specify the expected output. For that, he needs an oracle to define the correct resulting program or whether the transformation must be rejected.

For example, Listing 2.4 illustrates two test cases created by JRRT [68] developers to evaluate the Push Down Method refactoring implementation. The first test case (*test1*) contains an input program with classes *A*, *B* (subclass of *A*), and *C*. Class *A* has a method *m*. The expected output program contains the same classes *A*, *B*, and *C* but with the *m* method in the *B* class. After performing this test, if the engine produces an output different from the

expected one, the test fails.

Listing 2.4: Test cases of JRRT (02/03/13) to evaluate the Push Down Method refactoring.

---

```
1 public void test1 () {
2     testSucc (
3         Program.fromClasses (
4             "class A { void m() {} }",
5             "class B extends A {}",
6             "class C {}"),
7         Program.fromClasses (
8             "class A {}",
9             "class B extends A { void m() {} }",
10            "class C {}"));
11 }
12 public void test2 () {
13     testFail (
14         Program.fromClasses (
15             "class A { void m() {} }",
16             "class B extends A {}",
17             "class C { { new A().m(); } }"));
18 }
```

---

The second test case (*test2*) describes a situation in which the refactoring engine cannot apply the transformation. The input program has the same classes and methods of the input program of *test1* but with a call to the method *A.m()* from class *C*. If the engine applies the refactoring moving method *m* from class *A* to class *B*, the resulting program does not compile. Therefore, the refactoring should not be applied.

Manually writing test cases may be costly, and thus it may be difficult to create a good test suite considering combinations of all language constructs. Researchers have proposed a number of automated techniques for testing refactoring engines [10; 22; 77; 21]. They may automate four major steps of the testing process: (i) generating test inputs (except the technique proposed by Gligoric et al. [21]); (ii) applying the refactoring implementation; (iii) checking the output correctness; (iv) and classifying the detected failures into distinct bugs.

For example, to automate the test input generation, Gligoric et al. [22] propose UDITA (an extension of ASTGEN [10]), a Java-like language to write program generators so that

developers can generate programs as test inputs. They used UDITA to generate about 5,000 programs with up to 3 classes as test inputs [22]. Soares et al. [80; 77] propose a Java program generator called JDOLLY for exhaustively generating programs. By using JDOLLY, developers can specify the number of some Java constructs and constraints for the generated programs by using Alloy [27], a formal specification language. They used JDOLLY to generate more than 100,000 programs. Later, Gligoric et al. [21] propose real programs as test inputs, automatically applying the refactoring under test in every possible location of the program in which the refactoring under test can be applied. They found 141 bugs related to compilation errors and engine crash in 8 real systems in Java and C. In this work, we propose a technique to scale testing of refactoring engines.

There are several kinds of bugs in the refactoring implementations, such as bugs related to overly weak and overly strong conditions. Next, we explain those kinds of bugs.

### Overly Weak Conditions

Overly weak conditions cause the refactoring engine to apply transformations that do not compile or preserve the program behavior. Overly weak conditions can also lead to a system crash when the engine tries to apply a transformation. Soares et al. [77] catalogued a number of bugs related to overly weak conditions in refactoring implementations. Next, we show two of them.

Listings 2.5 and 2.6 show a transformation applied by Eclipse 3.7 that does not compile. Consider the class hierarchy presented in Listing 2.5. Classes *A* and *B* declare fields *f* and *n*, respectively. Class *C* declares *m* method, which accesses *f*. By using Eclipse JDT 3.7 to apply the Rename Field refactoring to change the name of *n* to *f*, results in the program presented in Listing 2.6. However, the resulting program does not compile. After the transformation, *B.f* hides *A.f*, and since the first one is private, it cannot be accessed from *C*. The following compilation error is introduced: “*The field B.f is not visible.*” The precondition should check if the new field can change the binding of a field call.

Figure 2.2.4 illustrates another example of bugs related to overly weak conditions. It shows a transformation applied by Eclipse 3.7 that does not preserve the program behavior. Consider class *A* and its subclass *B* as illustrated in Listing 2.7. Class *A* declares *k* method, and class *B* declares methods *k*, *m*, and *test*. The latter yields 1. Suppose we want to apply

Listing 2.5: Before refactoring.

---

```

public class A {
    int f = 1;
}
public class B extends A {
    private int n = 2;
}
public class C extends B {
    public int m() {
        return super.f;
    }
}

```

---

Listing 2.6: After refactoring: applying Rename Field in Eclipse JDT 3.7 leads to a compilation error due to field hiding.

---

```

public class A {
    int f = 1;
}
public class B extends A {
    private int f = 2;
}
public class C extends B {
    public int m() {
        return super.f;
    }
}

```

---

the Pull Up Method refactoring to move  $m$  from class  $B$  to class  $A$ . This method contains a reference to  $A.k$  using the *super* access. Performing this refactoring using Eclipse JDT 3.7 produces the program presented in Listing 2.8. The transformation updates the *super* qualifier to *this*. Then, a compilation error is avoided with this change. Nevertheless, a behavioral change was introduced: method *test* yields 2 instead of 1. Since  $m$  is invoked by an instance of  $B$ , its call is dispatched to  $B.k$  method implementation. The precondition should check whether changing a *super* modifier to *this* can change the dynamic binding of a method call.

### Overly Strong Conditions

Overly strong conditions cause the refactoring engine to reject applying a transformation that preserves the program behavior. For example, Listings 2.9 and 2.10 illustrate a behavioral preserving transformation rejected by JRRT [68] (02/03/13) due to overly strong conditions. Consider the class  $A$  and its subclass  $B$  presented in Listing 2.9. Class  $A$  declares method  $m(int)$  and class  $B$  declares method  $m(long)$ . Method  $B.test$  calls method  $m$  from an object of class  $B$ . As the methods  $A.m(int)$  and  $B.m(long)$  are overloaded,  $B.test$  calls  $A.m(int)$  to avoid using an implicit cast. By using JRRT to apply the Move Method refactoring to move

$A.m(int)$  to class  $B$ , it rejects this transformation. It reports the following warning message: Cannot adjust accessibilities.

Listing 2.7: Before refactoring.

---

```

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

```

---

Listing 2.8: After refactoring: applying Pull Up Method in Eclipse JDT 3.7 leads to a behavioral change due to incorrect change of **super** to **this**.

---

```

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

```

---

However, we can apply this transformation without changing the program behavior. Listing 2.10 illustrates a resulting program after removing a subset of overly strong conditions in JRRT that raises this warning message. Method *test* yields 0 in both versions of the program.

In our previous work [80], we proposed a technique to identify overly strong conditions in refactoring implementations based on differential testing. It needs at least two engines to apply the same kind of refactoring. If an engine rejects a transformation, and the other one applies it and preserves behavior according to SAFEREFACITOR [79], the technique establishes that the former engine contains an overly strong condition. In this work, we propose a new technique to identify overly strong conditions by detecting restrictive preconditions within the engine's code and turning them off (disabling them).



Listing 2.9: Original version.

---

```

public class A {
  public B f = null;
  protected long m(int b) {
    return 0;
  }
}
public class B extends A {
  long m(long b) {
    return 1;
  }
  public long test() {
    return B.this.m(2);
  }
}

```

---

Listing 2.10: Correct target's version after removing a subset of overly strong conditions.

---

```

public class A {
  public B f = null;
}
public class B extends A {
  long m(long b) {
    return 1;
  }
  protected long m(int b) {
    return 0;
  }
  public long test() {
    return B.this.m(2);
  }
}

```

---

## 2.3 Alloy Overview

Alloy is a formal specification language, based on first order logic that allows users to specify software systems by abstracting their key characteristics [27]. An Alloy specification is a sequence of signatures and constraints paragraphs declarations. A signature introduces a type and can declare a set of relations. Alloy relations have a multiplicity that is specified using qualifiers, such as *one* (exactly one), *lone* (zero or one), *set* (zero or more), and *seq* (sequence of elements). In Alloy, one signature can extend another, establishing that the extended signature is a subset of the parent signature. Next we specify a list of objects in Alloy. Each list (*List*) may have a sequence of objects (*Object*) in the relation *objs*.

---

```

sig Object {}
sig List {
  objs: seq Object
}

```

---

Facts are used to package formulas that always hold. The fact *listSize* specifies that all lists must have at most 10 elements.

---

```
fact listSize {
  all l: List | #l.objs <= 10
}
```

---

The keywords *all*, *some*, and *no* denote the universal, existential, and non-existential quantifiers, respectively. Predicates are used to package reusable formulas and specify operations. The predicate *noEmptyList* specifies that all lists are non-empty. The relation *objs* yields the elements of the sequence and the relation *isEmpty* checks whether a sequence is empty. The keyword *no*, when applied to an expression, denotes that the expression is empty.

---

```
pred noEmptyList[] {
  no l: List | l.objs.isEmpty
}
```

---

Moreover, we can declare functions (*fun*) in Alloy. For example, the function *getFirstElement* yields the first element of a list by using the helper function *first*.

---

```
fun getFirstElement [l: List] : one Object {
  l.objs.first
}
```

---

The predicate *addElement* adds an element in a sequence. For this purpose, the helper functions *add[o]* is used.

---

```
pred addElement [l: List, o: Object] {
  l.objs.add[o]
}
```

---

The Alloy Analyzer tool allows us to perform analysis on an Alloy specification [28]. A `run` command is applied to a predicate, specifying a scope for all declared signatures. For example, in the following command the Alloy Analyzer searches for an instance with at most three objects (scope) for *List* and *Object* satisfying all signature and fact constraints, in addition to the constraints specified in *noEmptyList*. The Alloy Analyzer has a feature to find all valid instances for a given scope.

---

```
run noEmptyList for 3
```

---

## 2.4 Technique to Test Refactoring Engines

In this section, we explain the technique to test refactoring engines [77; 80] that we extend in this work. First, Section 2.4.1 shows an overview of the technique. Next, Section 2.4.2 explains how to create the test inputs. Sections 2.4.3, 2.4.4, and 2.4.5 describe the process of refactoring application, oracles and bug categorizers.

### 2.4.1 Overview

First, the technique uses an automated program generator, DOLLY, to generate the test inputs (Step 1). It exhaustively generates Java and C programs. Next, the technique automatically applies the transformations under test using a refactoring engine (Step 2). It uses a set of automated oracles to evaluate the correctness of the transformations (Step 3). Finally, a set of bug categorizers is used for classifying the detected failures into distinct bugs (Step 4). Figure 2.1 illustrates the main steps of the technique.

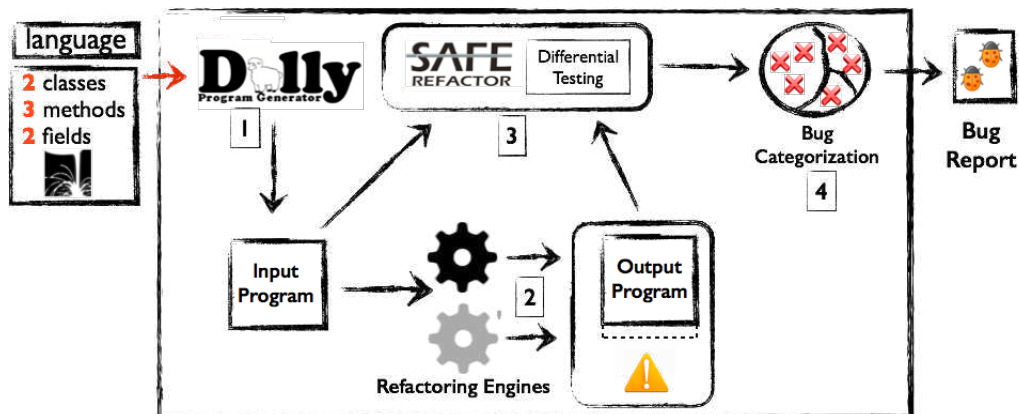


Figure 2.1: Technique for testing refactoring engines.

### 2.4.2 Test Input Generation

Manually creating test inputs for refactoring engines is costly and thus, time consuming, since developers need to provide complex inputs, such as programs. This may lead to a test suite with a low level of code coverage, potentially leaving many hidden faults. So, the technique uses DOLLY, an automated program generator to generate the test inputs. DOLLY

is a program generator that exhaustively generates programs, up to a given scope. DOLLY generates programs for Java (JDOLLY) and C (CDOLLY).

In this section, we explain JDOLLY. CDOLLY follows a similar approach. Next we present the Java meta-model used by JDOLLY (Section 2.4.2) and describe how to translate each Alloy instance to Java and how to use JDOLLY for generating more specific Java programs (Section 2.4.2).

### Java Meta-Model

Soares et al. [77] specified a Java meta-model in Alloy considering a subset of Java constructs. This Java meta-model is used by JDOLLY to generate Java programs. From Java, JDOLLY considers the primitive types *long* and *int*. A class is the only non-primitive type. A Java class has an identifier, field and method declarations, and can extend another class. Moreover, each class is located into 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, which can be *public*, *protected*, and *private*. A method declaration contains a return type, an identifier, a number of parameters, a body, and a modifier related to its accessibility. The methods can have at most one parameter and one statement in its body (a return statement). A return statement can have a *FieldAccess*, a *ConstructorFieldAccess*, a *MethodInvocation*, or a *LiteralValue*. A *MethodInvocation* and *FieldAccess* may contain a qualifier, such as *super* and *this*. Figure 2.2 illustrates the UML diagram representing the subset of the Java metamodel specified in Alloy.

The Alloy specification of JDOLLY also contains well-formedness rules within Alloy facts to reduce the rate of uncompileable programs. For example, the following fact specifies that no class can extend itself directly or indirectly. The operator  $\hat{\phantom{x}}$  represents the non-reflexive transitive closure.

---

```
fact noClassExtendsItself {
  no c:Class | c in c. $\hat{\text{extend}}$ 
}
```

---

There are some additional rules specified to cope with state explosion. For example, the predicate *optimization* does not allow non-primitive parameters. The well-formedness rules aim to reduce the state space of the Java meta-model used by JDOLLY. It intends to limit the

programs to some constructs of the language and scope in order to avoid state explosion of the Alloy instances. Similarly, there are other elements of Java's abstract syntax and other well-formedness rules.

```

pred optimization [] {
  all m: Method | (#m.param = 1) => (m.param in Int_ + Long_)
  ...
}

```

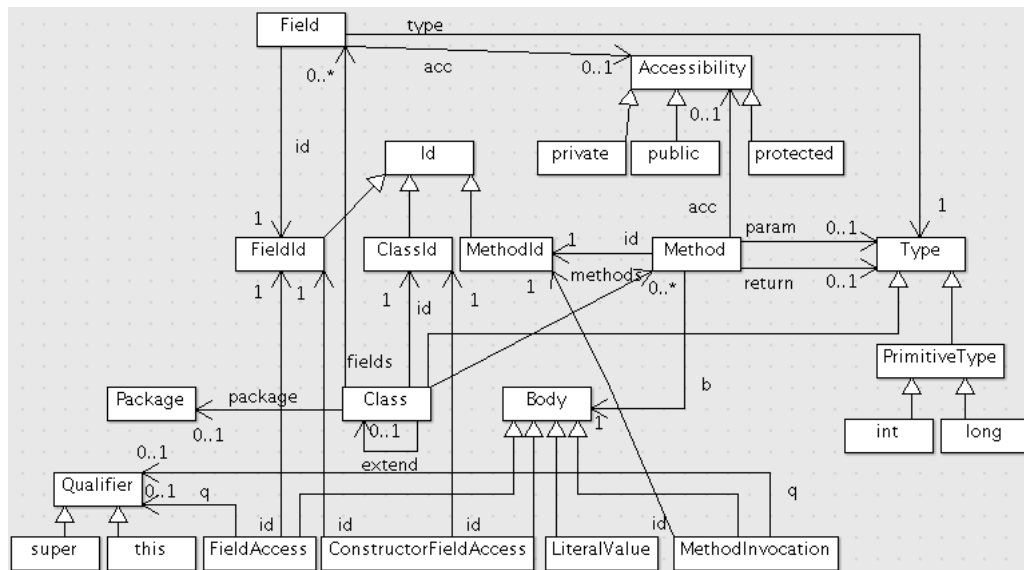


Figure 2.2: UML class diagram of JDOLLY's meta-model.

## Program Generation

JDOLLY uses this Alloy specification to generate Java programs. The specification contains a *run* command with a scope and a predicate to be satisfied. JDOLLY uses the Alloy Analyzer API to execute the run command for generating all solutions for the given scope. Each solution is an Alloy instance found by the Alloy Analyzer. DOLLY translates each Alloy instance into a Java program.

We can specify additional constraints to guide the program generation. For example, the following specification fragment states that the generated programs must have at least two classes and one field. The classes must be in the same direct hierarchy. The subclass must contain the field. Notice that the instances generated by the Alloy Analyzer may be used as input to test the Pull Up Field refactoring.

---

```
fact PullUpField {  
  some c1,c2:Class, f:Field | c1 in c2.extend && f in c2.fields  
}
```

---

### 2.4.3 Refactoring Application

The step of refactoring application consists of automatically applying the refactoring under test to each program generated as test input. For this purpose, the previous techniques [77; 80] implemented a program that uses the engine API to apply the refactorings automatically. The engine checks a set of preconditions before applying the transformation. If they are satisfied the transformation is applied. Otherwise, the transformation is rejected.

### 2.4.4 Test Oracle

After generating the test inputs and applying the refactorings, the technique must check whether the outputs were generated as expected. Refactoring engines can have overly weak and strong conditions [77; 80]. The technique uses a set of automated oracles to evaluate correctness of the transformations. It uses a compiler to identify bugs related to compilation errors, and SAFEREFACTOR [79] to identify behavioral changes. SAFEREFACTOR checks whether a transformation introduces behavioral changes. First, it analyzes the transformation to identify 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, the tool generates a test suite for those methods. Finally, it runs the tests before and after the transformation, and evaluates the results. If the results are different, the tool reports a behavioral change and displays the set of unsuccessful tests.

In our previous work, we proposed a technique [80] to identify overly strong conditions in refactoring implementations based on differential testing [43]. It needs at least two engines to apply the same kind of refactoring. If an engine rejects a transformation, and the other one applies it and preserves behavior according to SAFEREFACTOR, the technique establishes that the former engine contains an overly strong condition.

### 2.4.5 Bug Categorizer

In the previous step, the automated oracles may detect a number of failures in the refactoring implementations. A single bug in the refactoring may cause several of those failures. Next, we explain how to classify failures related to compilation errors, behavioral changes, and overly strong conditions.

#### Compilation Errors

Soares et al. [77] implemented a tool to categorize failures related to compilation errors and overly strong conditions. They use a similar technique proposed by Jagannath et al. [29] to automatically classify failures related to compilation errors into distinct bugs. It is based on splitting the failing tests based on messages from the test oracle (Oracle-based Test Clustering).

For instance, Listing 2.11 shows a program generated by DOLLY. If we apply the Rename Type refactoring by using Eclipse 3.7, the tool produces the output program shown in Listing 2.12, which contains the compilation error: “The hierarchy of the type *B* is inconsistent.” If another transformation results in an output program that contains the same compilation error message, even with another type name, the technique groups both transformations together by using the template of the compilation error: “The hierarchy of the type [*Type*] is inconsistent”. It ignores the parts inside quotes, which contain names of packages, classes, methods, and fields.

Listing 2.11: Original program.

```
package p1;
public class A extends B{}

package p1;
import p2.*;
public class B extends C {}

package p2;
public class C {}
```

Listing 2.12: Resulting program.

```
package p1;
public class C extends B{}

package p1;
import p2.*;
public class B extends C {}

package p2;
public class C {}
```

### Behavioral Changes

Soares et al. [77] specified a systematic, but manual approach to categorize failures related to behavioral changes based on structural characteristics of the transformation. The approach is based on a set of filters; a filter checks whether the programs follow a specific structural pattern, such as overloading, overriding, and implicit cast. All filters are presented in Table 2.1. The filters may be applied in any order. The fault category of a behavior-changing transformation is 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 by the refactoring engine developers.

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 <b>super</b> ( <b>this</b> or <b>implicit this</b> ) to <b>this</b> or <b>implicit this</b> ( <b>super</b> )	If a method call or field access has <b>this</b> or <b>implicit this</b> ( <b>super</b> ) as target, and after a refactoring this reference is replaced by <b>super</b> ( <b>this</b> or <b>implicit this</b> ), in order to keep the link to the same previous object
Maintains <b>super</b> while changing hierarchy	A reference to <b>super</b> 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

Table 2.1: Filters for classifying behavioral changes.

### Overly Strong Conditions

Manually analyzing each rejected behavior-preserving transformation to identify whether they show the same kind of overly strong condition is time consuming and error prone. In a previous work [80] we used a similar approach than the one used to classify the compilation errors failures, to automate the classification of failures related to overly strong conditions. For example, when we apply the Rename Method refactoring of Eclipse to the program



illustrated in Listing 2.13 the tool yields the following warning message: “Method *A.k(long)* will be shadowed by the renamed declaration *B.k(int)*”. The approach ignores the parts inside quotes, which contain names of packages, classes, methods, and fields. If there is another message reported by the same engine that has the same template, the rejected transformations are automatically classified in the same category of overly strong condition.

---

Listing 2.13: Renaming the *n* method to *k* is not allowed using Eclipse 3.7.

---

```
public class A {
    public long k(long a) {
        return 10;
    }
}

public class B extends A {
    public long n(int a) {
        return 20;
    }
    public long test() {
        return k(2);
    }
}
```

---

# Chapter 3

## SAFEREFACTORIMPACT

In this chapter, we present an overview of SAFEREFACTORIMPACT, the oracle used by our proposed technique to identify bugs related to behavioral changes in refactoring implementations. As we showed before, the refactoring engines can apply transformation that introduce behavioral changes. This problem is even worse with the presence of aspects (see Section 3.1). Previously, we proposed SAFEREFACTORIMPACT to evaluate transformations applied by refactoring engines in Java programs [45]. Here, we perform a new and more extensive evaluation in the context of Aspects to compare SAFEREFACTORIMPACT with SAFEREFACTOR [79] in new subjects considering OO and AO constructs, with respect to two new defined metrics (change coverage and relevant tests). Additionally, we combine SAFEREFACTORIMPACT to the oracles proposed to detect faults related to overly strong conditions (Chapters 4 and 6) and transformation issues (Chapter 5).

We organized this chapter as follows. Section 3.1 presents a motivating example. Next, Section 3.2 explains the technique of SAFEREFACTORIMPACT. Finally, Section 3.3 describes the evaluation of SAFEREFACTORIMPACT in 45 transformations applied in Java and AspectJ programs.

### 3.1 Motivating Example

In this section, we present a defective refactoring performed by Eclipse 4.2 with AJDT 2.2.3, which introduces a behavioral change. Consider classes *A*, *B*, *C*, and aspect *AspectA* presented in Listing 3.1. Class *C* extends *B*, which declares method *test*. *AspectA* declares

the method  $n$  in  $B$  through an inter-type declaration, which provides a way to express cross-cutting concerns affecting the structure of modules. By using Eclipse to apply the (aspect-aware) Rename Intertype Declaration refactoring to  $B.n$ , changing its name to  $B.k$ , we have as a result the program presented in Listing 3.2. Eclipse changed the intertype's name and updated its references. However, this transformation introduces a behavioral change: the  $test$  method in the target program now yields 20 (Listing 3.2) instead of 10 (Listing 3.1). After the transformation,  $test$  calls  $B.k$ , instead of the  $A.k$  method.

Listing 3.1: Original program.	Listing 3.2: Resulting program.
<pre> public class A {     public int k() {         return 10;     } }  public class B extends A {     public int test() {         return k();     } }  public class C extends B {     public int x() {         return 30;     } }  aspect AspectA {     public int B.n() {         return 20;     } } </pre>	<pre> public class A {     public int k() {         return 10;     } }  public class B extends A {     public int test() {         return k();     } }  public class C extends B {     public int x() {         return 30;     } }  aspect AspectA {     public int B.k() {         return 20;     } } </pre>

Figure 3.1: Applying the Rename Intertype Declaration refactoring of Eclipse 4.2 with AJDT 2.2.3 leads to a behavioral change.

Suppose that the developer has a test suite consisting of the test cases presented in Listing 3.3. It contains three test cases  $test1$ ,  $test2$ , and  $test3$  that call methods  $A.k$ ,  $B.test$ , and  $C.x$ , respectively. As explained, the transformation changed the behavior of method  $B.test$ .

Then, *test2* exposes the behavioral change in the resulting program. However, the other tests (*test1* and *test3*) are not relevant to test the transformation because the methods *A.k* and *C.x* are not impacted by the transformation.

Since only some test cases may be relevant to test the transformation, running all test cases may be a waste of time. Rachatasumrit and Kim [59] investigate the impact of refactoring edits on regression tests using the version history of Java open source projects. The results on three projects, JMeter, XMLSecurity, and ANT, show that existing regression tests exercise only 22% of refactored methods and fields and only 38% of tests are relevant to refactorings. In the previous example, the test suite only contains 33% of relevant tests. Furthermore, the tests may not exercise all entities impacted by the transformation. Therefore, to evaluate whether a transformation preserves the program behavior, it is more efficient to test only the methods impacted by the transformation.

Listing 3.3: Test suite of the program presented in Listing 3.1.

---

```
public void test1 () {
    A a = new A();
    long k = a.k();
    assertTrue(k == 10);
}
public void test2 () {
    B b = new B();
    long i = b.test();
    assertTrue(i == 10);
}
public void test3 () {
    C c = new C();
    long x = c.x();
    assertTrue(x == 30);
}
```

---

## 3.2 SafeRefactorImpact

SAFEREFACTORIMPACT uses change impact analysis to generate tests only for the entities impacted by the transformation. By comparing two versions of a program, it identifies the

methods impacted by the transformation (Step 1.1). We implemented a tool, called SAFIRA, to perform the change impact analysis, which identifies the public and common impacted methods in both program versions from the impacted set (Step 1.2). Next, SAFEREFACTORIMPACT generates a test suite for the previously identified impacted methods using an automatic test suite generator (Step 2). Since the tool focuses on identifying common methods, it executes the same test suite before (Step 3.1) and after the transformation (Step 3.2). Finally, the tool evaluates the results after executing the test cases: if the results are different, the tool reports a behavioral change, and yields the test cases that reveal it. Otherwise, we improve confidence that the transformation is behavior preserving (Step 4). Figure 3.2 illustrates the described process.

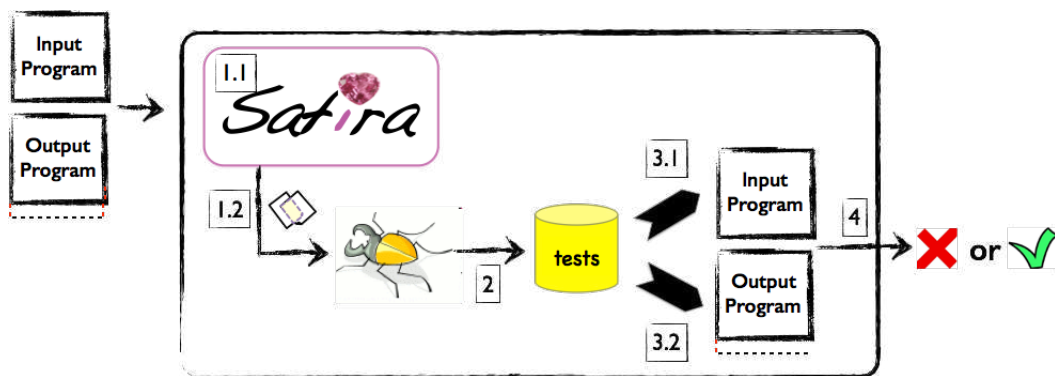


Figure 3.2: SAFEREFACTORIMPACT’s technique.

In what follows, we describe the change impact analysis (Section 3.2.1) and test generation steps (Section 3.2.2) of SAFEREFACTORIMPACT. Then, we describe a test data adequacy criteria [23] useful in the refactoring context, and define when a test case is relevant in Section 3.2.3.

### 3.2.1 Change Impact Analysis

In this section, we explain the change impact analysis performed by SAFEREFACTORIMPACT. The goal is to analyze the original and modified programs, and yield the set of methods impacted by the transformation. First, we decompose a coarse-grained transformation into smaller transformations. For each small-grained transformation, we identify the set of impacted methods. We formalized the impact of small-grained transformations in laws that specify the methods impacted by the transformation. Then, we identify the union of the set

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

### **Identifying Small-Grained Transformations**

We decompose the transformation into a set of small-grained transformations to analyze the impact of each one separately in the resulting program. We do this because it is simpler to analyze the impact of a small-grained transformation. Other change impact analyzers, such as Chianti [63] and FaultTracer [95], follow a similar approach but they depend on a test suite. They identify the tests cases impacted by a change. Our approach identifies the set of impacted methods.

To perform this step, we make a diff between the original and modified programs and identify the different kinds of transformations applied to the program. For example, if a transformation adds a method to a program, we consider it as the AM small-grained transformation. Another example is the CMB small-grained transformation, which modifies any part of a method body (adding, removing or changing a statement in a method body). Moreover, the CMM and CFM small-grained transformations add, remove or change a method and field modifier, respectively. Finally, the CFI and CSFI small-grained transformations add or remove field initializers or change the initialization value of instance and static fields, respectively. There may be other small-grained transformations not considered by our approach, such as changes in static blocks. Table 3.1 describes all small-grained transformations considered by our approach.

### **Identifying Impacted Methods**

After decomposing the coarse-grained transformation into smaller ones, we identify the impacted methods. We formalized the impact of each small-grained transformation described in Table 3.1. We grouped them into laws which define two small-grained transformations (from left to right and vice-versa), declaring two programs. The meta-variables *cds*, *fds* and *mds* define a set of class, field and method declarations, respectively. Each law specifies how we obtain the set of impacted methods when applying it in a particular direction.

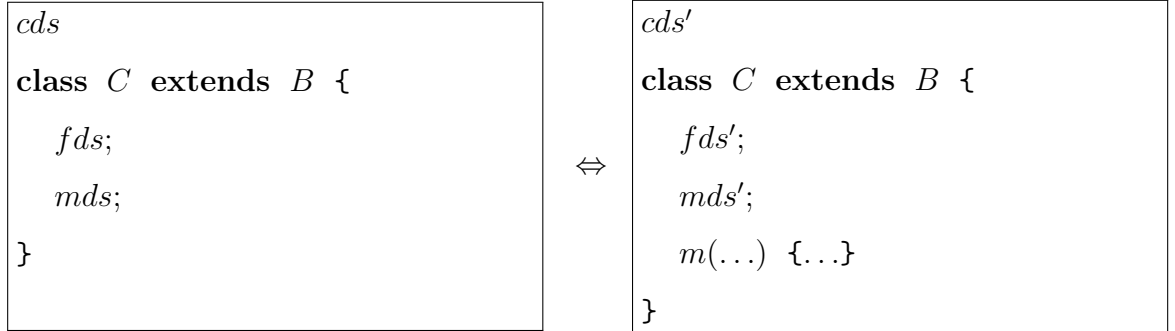
Table 3.1: Small-grained transformations considered by SAFIRA.

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

Next, we specify the impact of adding or removing a method. Law 1 adds the method  $m$  in the class  $C$  when applying it from left to right, and removes the method when applying it from right to left. The set of impacted methods is the same in both directions. We use  $\leftrightarrow$  to specify the impacted set for both directions. The transformation may change other program components but this law only identifies the impact of adding  $m$  method. If class  $B$  is *Object*, and  $C$  does not have a subclass, the set of impacted methods is  $C.m$ . Otherwise, other methods may be impacted due to overloading and overriding. For example, suppose that  $C$  has a superclass different than *Object* implementing  $m$ , and has a subclass  $D$  that does not implement  $m$ . Before the transformation,  $D.m$  resolves to  $B.m$ , but  $C.m$  is called after the transformation. So,  $D.m$  may change its behavior. We consider as impacted all methods that inherit  $m$  from  $C$ . We denote the subclass relation by  $<$ .

We specified other laws for the small-grained transformations presented in Table 3.1 similarly. After decomposing the coarse-grained transformation into smaller ones, we identify the impacted methods of each of them using our laws. The set of directly impacted methods is the union of the impacted set of each small-grained transformation. After that, we also identify the set of indirectly impacted methods that exercise an impacted method directly or indirectly. Finally, the union of directly and indirectly impacted methods defines the resulting set of impacted methods.

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

**Law 1** ⟨Add/Remove Method⟩

$(\Leftrightarrow) \{n:\text{Method} \mid \exists E:\text{Class} \cdot (F < E \wedge E \leq C) \wedge (n \in \text{methods}(c_{ds}') \cup m_{ds}') \wedge n = E.m\}$ ,

where  $F$  is the closest subclass of  $C$  such that overrides  $m$ .

uses ASM,<sup>1</sup> a framework to analyze and manipulate Java bytecode, to identify small-grained transformations and methods impacted. Since the tool analyzes Java bytecode and the AspectJ compiler translates an AspectJ program to Java bytecode, we do not specify laws for AspectJ constructs. This way, we can analyze the transformation using Safira.

### 3.2.2 Test Generation

From the impacted methods set identified by SAFIRA, we identify the public and common methods in both program versions. We pass them to an automatic test suite generator. Finally, we execute the same generated test suite before and after the transformation. If the results are different, we show a test case exposing the behavioral change. Otherwise, we improve confidence that the transformation is behavior preserving. SAFEREFACTORIMPACT uses Randoop [65; 58] to automatically generate a test suite for the methods impacted by the transformation. Randoop can receive as parameter a set of methods and randomly generates unit tests for these methods within a time limit.

### 3.2.3 Change Coverage and Relevant Tests

Rachatasumrit and Kim [59] provide evidence that refactorings are not well tested. They found that existing regression test suites may not cover the impacted entities, and a num-

<sup>1</sup><http://asm.ow2.org/>



ber of test cases may not be relevant for testing the refactorings. Based on their work, we define two metrics for evaluating the test suites generated by SAFEREFACTORIMPACT and SAFEREFACTOR: Change Coverage and Relevant Tests.

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

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

Considering the transformation presented in Figure 3.1, suppose that the test suite consists of the test cases presented in Listings 3.4 and 3.5. The first test case calls method *B.test*, that calls *A.k* in the original program and *B.k* in the modified one. The second test case calls the method *C.x*. The set of impacted methods by this transformation is: *B.k*, *C.k*, *B.n*, *C.n*, *B.test* and *C.test*. The test suite exercises two out of six impacted methods. So, the change coverage is:  $C = \frac{2}{6} = 33\%$ . Since the second test case does not exercise any impacted method, it is not relevant. So, the percentage of relevant tests in this example is:  $R = \frac{1}{2} = 50\%$ . Notice that some impacted methods do not belong to both programs, such as *B.n* and *C.n*, and they are not called by other methods. Sometimes it is not possible to generate tests for them since SAFEREFACTORIMPACT generates a test suite that must execute in both versions of the program.

Listing 3.4: A unit test revealing a behavioral change in the transformation presented in Figure 3.1.

---

```

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

```

---

Listing 3.5: A non-relevant unit test generated by Randoop in SAFEREFACTOR used to

---

evaluate the transformation presented in Figure 3.1.

---

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

---

## 3.3 Evaluation

In this section, we present our experiment to compare two approaches for identifying behavior-preserving transformations. First, we present the experiment definition and planning. Then, we present the results and discussion. Next, we describe some threats to validity. Finally, we summarize the main findings. All tools and experimental data are available online.<sup>2</sup>

### 3.3.1 Definition

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

- **Q1.** Do SAFEREFACTORIMPACT and SAFEREFACTOR detect the same behavioral changes?

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

- **Q2.** Is SAFEREFACTORIMPACT faster than SAFEREFACTOR to evaluate a transformation?

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

---

<sup>2</sup>[http://www.dsc.ufcg.edu.br/~spg/mongiovi\\_thesis.html](http://www.dsc.ufcg.edu.br/~spg/mongiovi_thesis.html)

- **Q3.** Does SAFEREFACTORIMPACT generate a test suite with better change coverage than SAFEREFACTOR?

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

- **Q4.** Does SAFEREFACTORIMPACT use a test suite to evaluate a transformation with more relevant test cases than SAFEREFACTOR?

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

### 3.3.2 Planning

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

#### 3.3.2.1 Hypothesis formulation

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

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

$$H_0 : Time_{SRI} \geq Time_{SR} \quad (3.1)$$

$$H_1 : Time_{SRI} < Time_{SR} \quad (3.2)$$

- To answer **Q3**, concerning the change coverage of the generated tests:

$$H_0 : ChangeCoverage_{SRI} \leq ChangeCoverage_{SR} \quad (3.3)$$

$$H_1 : ChangeCoverage_{SRI} > ChangeCoverage_{SR} \quad (3.4)$$

- To answer **Q4**, concerning the percentage of relevant tests:

$$H_0 : RelevantTests_{SRI} \leq RelevantTests_{SR} \quad (3.5)$$

$$H_1 : RelevantTests_{SRI} > RelevantTests_{SR} \quad (3.6)$$

We perform statistical analysis for each group of subjects that contains at least eight transformations. We use Shapiro-test [75] to analyze data normality because it is more adequate for small samples. Then, if the data are normal, we use T-test [7], otherwise we use Wilcoxon-test [89]. We use the level of significance 0.5.

### **Selection of subjects**

We evaluated SAFEREFACTOR and SAFEREFACTORIMPACT in eight defective refactorings applied by Eclipse 4.2 using AJDT 2.2.3 that introduce behavioral changes in AO programs, 23 design patterns implemented in Java and AspectJ [25], in two programs compiled by two Java Modeling Language (JML) [37] compilers [61; 60], and 12 transformations applied to real OO and AO programs (JHotDraw and CheckStylePlugin 4.2). We selected these subjects in order to evaluate the tools in transformations with different granularities and applied to programs with different sizes and constructs. Experienced developers and researchers in the OO and AO field applied the transformations, which have different granularities, to programs with different sizes (ranging from 10 LOC to 79 KLOC). The transformations change OO (classes, methods, fields, inheritance, overloading, overriding, packages, accessibility) and AO (aspects, intertype declarations, pointcuts, advices) constructs. We analyzed local and global transformations. Some of them affect classes, aspects and method signatures, while others change blocks of code only within methods.

### **Experiment design**

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

### **Instrumentation**

We ran the experiment on a 2.7 GHz core i5 with 8 GB RAM and running Mac OS 10.8. We used the command line interfaces of SAFEREFACTOR 1.1.4 and SAFEREFACTORIMPACT 1.0 using Java 1.6. They receive as parameters the original and the target program paths, and

the time limit to generate tests. We used SAFIRA 1.0, which uses ASM 3.0. We used a time limit of 0.2s, 0.5s and 0.2s to generate tests for the defective refactorings, design patterns, and bytecode generated by two JML compilers, respectively. These limits are enough to test transformations applied to small programs [77]. We used a time limit of 20s for the larger case studies (up to 79 KLOC). Both tools use Randoop 1.3.3, configured to avoid generating non-deterministic test cases. Since we do not know beforehand which versions contain behavior-preserving transformations, we compared the results of all approaches in all transformations to establish a Baseline to check the results of each approach. For instance, if SAFEREFACTOR yielded “Yes” and SAFEREFACTORIMPACT “No”, we checked whether the test case showing the behavioral change reported by SAFEREFACTORIMPACT was correct. If so, the correct result was “No”.

### 3.3.3 Results

Next, we summarize the main results of our evaluation in the defective refactorings, design patterns, bytecode generated by two JML compilers, and larger case studies.

#### Defective Refactorings

SAFEREFACTOR and SAFEREFACTORIMPACT correctly identified all behavioral changes but one, Subject 2, that only SAFEREFACTOR identified. As expected, SAFEREFACTORIMPACT evaluated the subjects faster than SAFEREFACTOR. Furthermore, both tools had almost the same change coverage in all subjects but Subject 4 in which SAFEREFACTORIMPACT had 100% of change coverage and SAFEREFACTOR 25%. Finally, all test cases generated by SAFEREFACTORIMPACT are relevant to test the change different from SAFEREFACTOR. Table 3.2 summarizes the results.

#### Design Patterns

SAFEREFACTORIMPACT correctly identified behavioral changes in 5 out of 23 the design patterns implementations [25]. SAFEREFACTOR identified all of these behavioral changes but one (the Mediator design pattern), which can only be detected using a time limit of three seconds. Hannemann and Kiczales [25], which applied these transformations, did not expect

Table 3.2: Results using a time limit of 0.2s. Methods = number of methods passed to Randoop to generate tests; Time = the total time of the analysis in seconds; Change Coverage = the percentage of impacted methods covered; Relevant Tests = the percentage of relevant tests; Result = it states whether the transformation is behavior-preserving.

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

to introduce behavioral changes in none of them. However, we found behavioral changes in the Mediator, Prototype, State, Template and Visitor design patterns. SAFEREFACTORIMPACT evaluated the subjects faster than SAFEREFACTOR. Both tools have almost the same change coverage except for Subjects 9, 18 and 30, but SAFEREFACTORIMPACT generated more relevant tests than SAFEREFACTOR. Table 3.3 summarizes the results.

### JML Compiler

SAFEREFACTOR and SAFEREFACTORIMPACT correctly identified behavioral changes in both transformations (Subjects 32 and 33). Both tools took almost the same time to evaluate Subject 32. However, SAFEREFACTORIMPACT took more time to evaluate Subject 33 since the change impact analysis is more expensive because the transformation impacted more than 6,000 methods. Table 3.4 summarizes the results. Moreover, both tools had similar low change coverage. Finally, SAFEREFACTORIMPACT generated more relevant test cases than SAFEREFACTOR.

### Larger Case Studies

SAFEREFACTORIMPACT correctly evaluated all transformations but two (Subjects 38 and 39), while SAFEREFACTOR correctly evaluated seven transformations. Both tools took almost the same time to evaluate the subjects. As expected, SAFEREFACTORIMPACT had

Table 3.3: Results using a time limit of 0.5s. Impacted Methods = number of methods identified by SAFIRA; Methods = number of methods passed to Randoop to generate tests; Time = the total time of the analysis in seconds; Change Coverage = the percentage of impacted methods covered; Relevant Tests = the percentage of relevant tests; Result = it states whether the transformation is behavior preserving.

Subject	Design Pattern	Impacted Methods	Methods		Time (s)		Change Coverage (%)		Relevant Tests (%)		Result		Baseline
			SR	SRI	SR	SRI	SR	SRI	SR	SRI	SR	SRI	
9	Abstract Factory	107	315	10	16	4	2	7	33.33	100	Yes	Yes	Yes
10	Adapter	95	4	4	8	3	1	1	97.56	100	Yes	Yes	Yes
11	Bridge	122	14	16	8	3	12	14	97.05	98.86	Yes	Yes	Yes
12	Builder	115	11	12	12	3	11	13	98.87	100	Yes	Yes	Yes
13	C. of Responsibility	130	998	7	18	3	0	0	0	0	Yes	Yes	Yes
14	Command	116	384	5	13	6	6	6	9.09	100	Yes	Yes	Yes
15	Composite	131	7	7	9	5	2	2	69.52	91.08	Yes	Yes	Yes
16	Decorator	108	4	4	10	4	2	2	97.67	100	Yes	Yes	Yes
17	Facade	96	14	14	8	3	8	8	95.29	100	Yes	Yes	Yes
18	Factory Method	113	8	10	15	7	7	12	91.66	100	Yes	Yes	Yes
19	Flyweight	99	6	6	10	3	2	2	98.18	100	Yes	Yes	Yes
20	Interpreter	135	38	27	8	3	6	9	66.67	84.62	Yes	Yes	Yes
21	Iterator	109	7	7	10	3	3	3	100	100	Yes	Yes	Yes
22	Mediator	99	714	5	17	3	0	3	0	50.00	Yes	No	No
23	Memento	97	5	5	8	3	2	2	98.36	100	Yes	Yes	Yes
24	Observer	134	11	11	8	5	2	2	96.67	100	Yes	Yes	Yes
25	Prototype	103	10	10	9	3	5	5	88.61	88.78	No	No	No
26	Proxy	144	6	6	10	3	6	6	100	100	Yes	Yes	Yes
27	Singleton	115	1	2	10	3	2	5	100	100	Yes	Yes	Yes
28	State	123	9	9	9	3	21	20	96.67	100	No	No	No
29	Strategy	107	6	6	9	3	1	1	90.91	100	Yes	Yes	Yes
30	Template Method	104	10	11	12	3	8	15	98.73	100	No	No	No
31	Visitor	115	11	11	9	4	4	4	88.89	91.67	No	No	No

Table 3.4: Results using a time limit of 0.2s. Impacted Methods = number of methods identified by SAFIRA; Methods = number of methods passed to Randoop to generate tests; Time = the total time of the analysis in seconds; Change Coverage = the percentage of impacted methods covered; Relevant Tests = the percentage of relevant tests; Result = it states whether the transformation is behavior preserving.

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

higher percentage of change coverage in nine subjects, since it focuses on testing the methods impacted by the change. In the other three subjects, they had almost the same change coverage. SAFEREFACTORIMPACT generated at least 95% of relevant tests. In Subjects 39, 43 and 44, SAFEREFACTOR generated less than 10% of relevant tests since it passed more than 30,000 methods to Randoop generate tests. Table 3.5 summarizes the results.

Table 3.5: Results using a time limit of 20s. Impacted Methods = number of methods identified by SAFIRA; Methods = number of methods passed to Randoop to generate tests; Time = the total time of the analysis in seconds; Change Coverage = the percentage of impacted methods covered; Relevant Tests = the percentage of relevant tests; Result = it states whether the transformation is behavior preserving.

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

### 3.3.4 Discussion

Next, we discuss the results of our evaluation in the defective refactorings, design patterns, bytecode generated by two JML compilers, and larger case studies. We perform the statistical analysis for each group of subjects, since we used different time limits. Therefore, we cannot compare their results.

#### Defective Refactorings

SAFEREFACTORIMPACT does not detect the behavioral change in Subject 2, since SAFIRA does not perform data flow analysis. The original program is presented in Listing 3.6. It contains class *A* that declares field *x* and methods *getX*, *setX*, *m* (which calls *setX*) and *test*



(which calls  $m$ ). This program also contains the aspect *Update* that changes the value passed to  $setX$  when it is called by  $A.k(int)$  (this method does not exist in this program version). The transformation renames  $A.m$  to  $k$ . The resulting program is illustrated in Listing 3.7. The behavioral change is detected by a test case that calls  $A.test(a); A.getX$  ( $a \neq 1$ ). Since the aspect only changes the value passed to  $setX$  in the modified program,  $getX$  returns different values in the original and modified programs after a call to  $A.test$ . SAFEREFACTORIMPACT did not identify this behavioral change because it did not generate tests for  $getX$ . It has a parameter that, when enabled, allows us to consider all getter methods during the test suite generation. By enabling this parameter, SAFEREFACTORIMPACT correctly identifies the behavioral change in Subject 2. However, when using such option, the number of methods passed to the test suite generator may increase in some transformations.

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

Table 3.6 describes the statistical analysis results. We considered all data to run the tests. Column Shapiro Test indicates the Shapiro–Wilk test results. When we ran Shapiro-test in relevant tests data of SAFEREFACTORIMPACT an error occurred, because all data are equal (100% of relevant tests). Then, we consider it as non-normal. Notice that only the change coverage data of SAFEREFACTOR and SAFEREFACTORIMPACT are normal. Columns T-test and Wilcoxon-test present the results of the tests to evaluate the hypotheses presented in Section 3.3.2.1.

Due to non-normality of data, we use Wilcoxon-test for time, and percentage of relevant tests data. It reached small p-values to all of them:  $2.3 \times 10^{-4}$  and  $2.0 \times 10^{-4}$ , respectively. The results give us evidence that SAFEREFACTORIMPACT reduces time and generates more

Listing 3.6: Original program.

---

```

public class A {
    int x;
    public void test(int x) {
        m(x);
    }
    public void m(int x) {
        setX(x);
    }
    public void setX(int x) {
        this x = x;
    }
    public int getX() {
        return this.x;
    }
}

public aspect Update {
    void around(int x) : call(
        void A.k(..) ) && args(x) {
        x = 1;
        proceed(x);
    }
}

```

---

Listing 3.7: Resulting program after the transformation applied in Subject 2.

---

```

public class A {
    int x;
    public void test(int x) {
        k(x);
    }
    public void k(int x) {
        setX(x);
    }
    public void setX(int x) {
        this x = x;
    }
    public int getX() {
        return this.x;
    }
}

public aspect Update {
    void around(int x) : call(
        void A.k(..) ) && args(x) {
        x = 1;
        proceed(x);
    }
}

```

---

relevant tests than SAFEREFACOR for these subjects. To evaluate change coverage we use T-test due to normality of data. It reached a p-value of 0.25 which indicates that the change coverage of SAFEREFACORIMPACT is less than or similar to the change coverage of SAFEREFACOR. Then, we execute another test (T-test) assuming a null hypothesis that the change coverage is equal for both tools. It reached a p-value of 0.51, which indicates that there is no statistical difference between the change coverage of SAFEREFACOR and SAFEREFACORIMPACT.

Table 3.6: Statistical analysis for defective refactoring data. Shapiro Test = analyze data normality; T-test = evaluate hypothesis test when data are normal; Wilcoxon-test = evaluate hypothesis test when data are non-normal; Result = final results of the statistical analysis; SR = SAFEREFACTOR; SRI = SAFEREFACTORIMPACT.

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

## Design Patterns

SAFEREFACTORIMPACT found a behavioral change in the Mediator pattern implementations that SAFEREFACTOR cannot find using the same time limit. Developers implemented a GUI application and used the mediator pattern to deal with changes to GUI components that require updates. In the OO version, they implemented this pattern as a field of the component, which must be set by using a setter method. Notice that SAFEREFACTOR cannot detect this behavioral change. The time limit of 0.5s passed to Randoop is not enough to generate tests considering 714 methods. So, it did not generate relevant test cases and cover the change different from SAFEREFACTORIMPACT. Also, both tools found simple behavioral changes in three design patterns (Prototype, Template Method, and Visitor). Some methods yield different `String` messages.

Both tools had low change coverage. The number of impacted methods (see Column Impacted Methods in Table 3.3) identified by SAFIRA is larger (90%) than the number of methods passed to Randoop by SAFEREFACTORIMPACT. The transformation adds or removes most impacted methods. SAFEREFACTORIMPACT cannot pass them to Randoop because they do not belong to both versions of the program. As mentioned, our goal is to generate a test suite to be executed before and after the transformation. Furthermore, some methods contain parameter types declared in external libraries, such as Java AWT, in some subjects. Randoop does not generate test inputs for them unless we pass them as parameters, or some method being tested yields an object of the library's type. In the Chain of Responsibility implementations, all test cases generated by SAFEREFACTORIMPACT throw exceptions before executing the impacted method. We may increase the time limit, or this may indicate

a limitation of the test suite generator that cannot handle some kinds of Java constructions, such as GUI elements. So, the tool does not generate relevant tests to exercise the change in this subject. In Subjects Chain of Responsibility and Mediator, a similar scenario happens in SAFEREFACTOR. In some subjects, SAFEREFACTORIMPACT passed more methods in common to Randoop than SAFEREFACTOR. Different from SAFEREFACTOR, SAFEREFACTORIMPACT takes into consideration methods that are moved from a class to an aspect, that introduces it in the same class using an intertype declaration.

Table 3.7 describes the statistical analysis results. Column Shapiro Test consists of Shapiro-Wilk test results. The results indicate that all data are non-normal. Then, we use Wilcoxon-test for all of them. Column Wilcoxon-test presents the results of the test to evaluate the hypothesis presented in Section 3.3.2.1.

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

Table 3.7: Statistical analysis for design patterns data. Shapiro Test = analyze data normality; Wilcoxon-test = evaluate hypothesis test when data are non-normal; Result = final results of the statistical analysis; SR = SAFEREFACTOR; SRI = SAFEREFACTORIMPACT.

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

### JML Compiler

Different from what Rêbello et al. [61] expected, the programs compiled using the standard JML compiler (*jmlc*) and *ajmlc* are not equivalent. They must check invariants after creating

an object, and before and after a method call. By analyzing the tests reported by our tools, we detected that *ajmlc* checks invariants before each constructor. This led to false invariant violation warnings, which represents a bug in the *ajmlc*. The tools also detected a behavioral change during a postcondition evaluation of a method declared in *JAccount*.

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

### Larger Case Studies

In Subject 35, we evaluated the OO' and AO versions, and both tools also detected this behavioral change. SAFEREFACITOR and SAFEREFACITORIMPACT detected behavioral changes between the OO and AO versions of *CheckStylePlugin* (Subjects 36 and 37) using a time limit of 20s. In Subjects 38 and 39, both tools did not identify behavioral changes using a time limit of 20s. Randoop does not generate tests that exercise the impacted methods that change behavior using this time limit. Different from SAFEREFACITOR, SAFEREFACITORIMPACT identified the behavioral changes in both subjects using a time limit of 120s, since it reduces by more than 90% the number of methods passed to Randoop to generate tests.

Soares et al. [78], evaluated Subjects 40-45 using SAFEREFACITOR and a manual inspection performed by experts [50]. SAFEREFACITOR did not identify the behavioral changes using a time limit of 20s in Subjects 40, 42, 43, 44 and 45 different from SAFEREFACITORIMPACT. However, it detected three of them (Subjects 40, 43 and 45) using a time limit of 120s. Both manual inspection [50] and SAFEREFACITOR classified Subject 42 as behavior preserving. However, SAFEREFACITORIMPACT identified a previously undetected behavioral change in Subject 42. SAFEREFACITOR did not identify this behavioral change because Randoop does not generate tests to expose them using the time limit of 120s, since the number of methods to test is much greater (90%) than in SAFEREFACITORIMPACT. It is also important to notice that finding behavioral changes is not an easy task, even when using a well-defined manual inspection conducted by experts [50; 51].

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

Table 3.8 describes the statistical analysis results. Column Shapiro Test indicates the Shapiro–Wilk test results. Notice that only the change coverage data of SAFEREFACITOR and SAFEREFACITORIMPACT are normal. Columns T-test and Wilcoxon-test present the results of the tests to evaluate the hypothesis presented in Section 3.3.2.1.

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

### 3.3.5 Threats to Validity

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

Table 3.8: Statistical analysis for larger subjects data. Shapiro Test = analyze data normality; T-test = evaluate hypothesis test when data are normal; Wilcoxon-test = evaluate hypothesis test when data are non-normal; Result = final result of the statistical analysis; SR = SAFEREFACTOR; SRI = SAFEREFACTORIMPACT.

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

### Construct validity

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

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

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

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

### **Internal validity**

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

Finally, compilers may have introduced behavioral changes during the optimization process [93]. Since SAFEREFACTORIMPACT analyzes the Java bytecode, this may have an influence on the results if the compilers have bugs.

### **External validity**

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

Randoop does not deal with concurrency. In those situations, SAFEREFACTOR and SAFEREFACTORIMPACT may yield non-deterministic results. Also, they do not take into account characteristics of some specific domains. For instance, currently, they do not detect the difference in the standard output (*System.out.println*) message. Neither could the tool generate tests that exercise some changes related to the graphical interface (GUI) of JHotDraw. We can use another test generator to generate test cases from a set of methods.

### **Conclusion validity**

We use statistical tests to evaluate small samples (up to 23 data), which can reduce the power of the tests. This is a threat because low statistical power increases the likelihood of making a Type II error (accepting null hypothesis when it is false). To minimize this threat, we met the assumptions underlying the tests (e.g., normality) and choose T-test when the data come



from a population, that has a normal distribution, and Wilcoxon-test otherwise.

### 3.3.6 Answers to the Research Questions

From the evaluation results, we make the following observations:

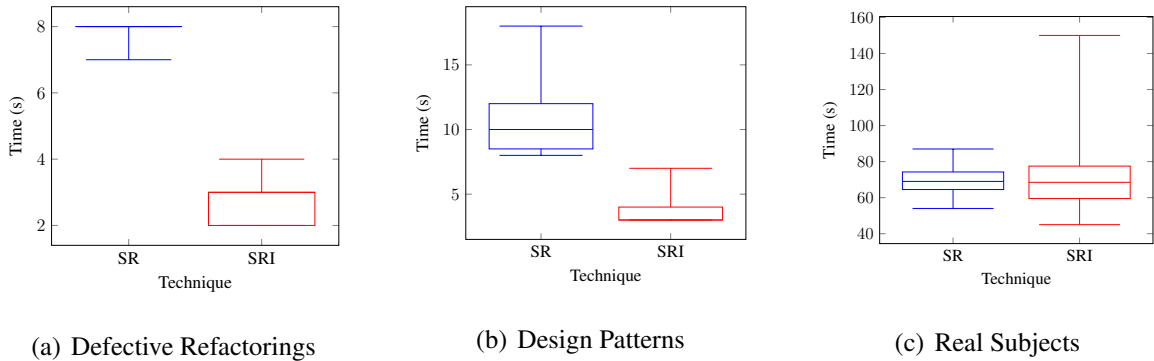
- **Q1.** Do SAFEREFACTORIMPACT and SAFEREFACTOR detect the same behavioral changes?

No. SAFEREFACTORIMPACT does not identify the behavioral change in Subject 2 due to a limitation in SAFIRA. If we pass all getter methods as parameter in the test generation, SAFEREFACTORIMPACT detects it. Moreover, it does not detect behavioral changes in Subjects 38 and 39 using a time limit of 20s. If we increase the time limit to 120s, it detects the behavioral change different from SAFEREFACTOR. On the other hand, SAFEREFACTORIMPACT detects behavioral changes in Subjects 22, 40, 42, 43, 44 and 45 that SAFEREFACTOR does not identify them using a time limit of 20s. SAFEREFACTOR detects the behavioral changes in Subjects 40, 43 and 45 using a time limit of 120s. SAFEREFACTORIMPACT finds a behavioral change in Subject 42 undetected by SAFEREFACTOR and a well-defined manual inspection conducted by experts. We calculate the precision and recall metrics for SAFEREFACTORIMPACT and SAFEREFACTOR. SAFEREFACTORIMPACT has a precision of 0.86 while SAFEREFACTOR has a precision of 0.7. As expected, the precision of SAFEREFACTORIMPACT is higher than the precision of SAFEREFACTOR, since SAFEREFACTORIMPACT identifies less false-positives. Both tools have a recall of 1, since they do not have false-negatives.

- **Q2.** Is SAFEREFACTORIMPACT faster than SAFEREFACTOR to evaluate a transformation?

In the transformations applied to small programs, SAFEREFACTORIMPACT is faster than SAFEREFACTOR. However, both tools take almost the same time to evaluate transformations applied to larger programs. Figure 3.3 illustrates the distribution of the total time to evaluate transformations by SAFEREFACTOR and SAFEREFACTORIMPACT in the subjects of defective refactorings, designs patterns, and larger case studies.

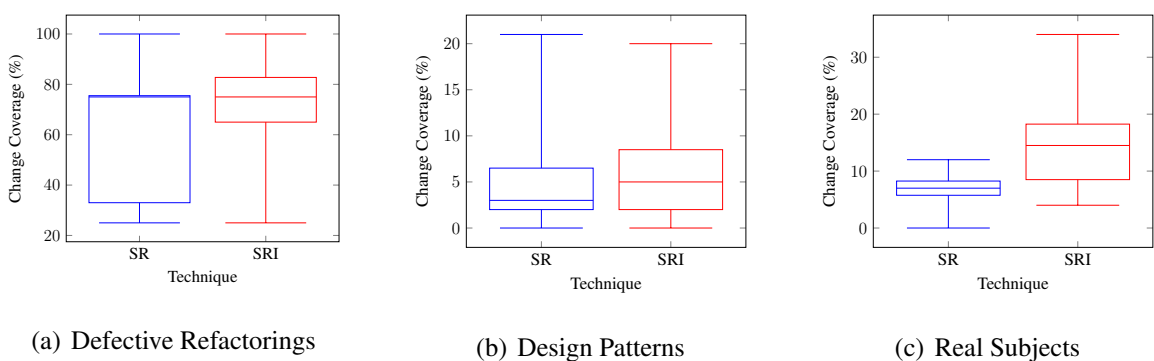
Figure 3.3: Distribution of the total time to evaluate transformations by SAFEREFACITOR and SAFEREFACITORIMPACT.



- **Q3.** Does SAFEREFACITORIMPACT generate a test suite with better change coverage than SAFEREFACITOR?

The test cases generated by SAFEREFACITORIMPACT increase the change coverage in larger subjects. For small ones, there is no significant difference, but in most of the subjects, it is similar or better than SAFEREFACITOR. Figure 3.4 illustrates the distribution of the change coverage of the tests generated by SAFEREFACITOR and SAFEREFACITORIMPACT in the subjects of defective refactorings, designs patterns, and larger case studies.

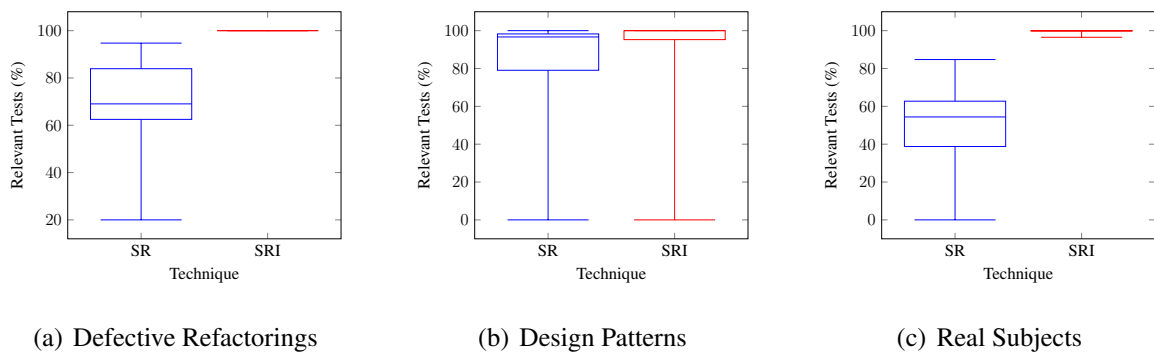
Figure 3.4: Distribution of the change coverage of the tests generated by SAFEREFACITOR and SAFEREFACITORIMPACT.



- **Q4.** Does SAFEREFACITORIMPACT use a test suite to evaluate a transformation with more relevant test cases than SAFEREFACITOR?

Yes. SAFEREFACTORIMPACT generates more relevant tests in all subjects. Almost 90% of test cases generated by SAFEREFACTORIMPACT are relevant to evaluate the change. Some test cases are not relevant because they throw an exception before or while executing an impacted method. Figure 3.5 illustrates the distribution of the percentage of relevant tests generated by SAFEREFACTOR and SAFEREFACTORIMPACT in the subjects of defective refactorings, designs patterns, and larger case studies.

Figure 3.5: Distribution of the percentage of relevant tests generated by SAFEREFACTOR and SAFEREFACTORIMPACT.



# Chapter 4

## A Technique to Scale Testing of Refactoring Engines

In this chapter we present our technique to scale testing of refactoring engines by extending a previous technique [77; 80]. The previous technique uses DOLLY to generate programs as test inputs and contains a set of oracles and bug categorizers to identify bugs related to compilation errors, behavioral changes, and overly strong preconditions. Our goal is to scale it by improving expressiveness of the program generator, reducing costs, improving performance and bug detection, and also detecting more bug types.

First, Section 4.1 presents an overview of the technique. Next, Section 4.2 explains the new features added in DOLLY. Finally, Section 4.3 describes the evaluation of the technique using skips to generate programs with respect to overly weak preconditions (compilation errors and behavioral changes) and overly strong preconditions using Differential Testing Technique (DT technique) [80].

### 4.1 Overview

In this section, we explain the main steps of our technique. First, it automatically generates programs as test inputs for a refactoring using DOLLY, an automated program generator (Step 1). DOLLY receives as input the refactoring type to be tested, a skip number to reduce the number of generated programs, and an Alloy specification, which includes specific constraints to a refactoring type and the scope of the programs. DOLLY can generate Java

(JDOLLY) or C (CDOLLY) programs. Next, the refactoring under test is automatically applied to each generated program. When there are more than one refactoring engine, the technique uses all of them to apply the transformation (Step 2). To evaluate the correctness of the transformations, our technique uses a set of oracles that can identify overly weak preconditions and transformation issues. It also contains an oracle to evaluate overly strong preconditions (Step 3). Finally, the detected failures are automatically categorized into distinct bugs (Step 4). Figure 6.1 illustrates the main steps.

Bugs related to compilation errors and behavioral changes occur when the refactoring engine has overly weak preconditions, which allow applying transformations that do not compile or preserve the program behavior. On the other hand, overly strong preconditions prevent the refactoring engine from applying safe transformations. When a transformation compiles and preserves the program behavior, but do not follow its refactoring definition, it may be a bug in the refactoring engine or a bad smell introduced in the program. We call them as transformation issues.

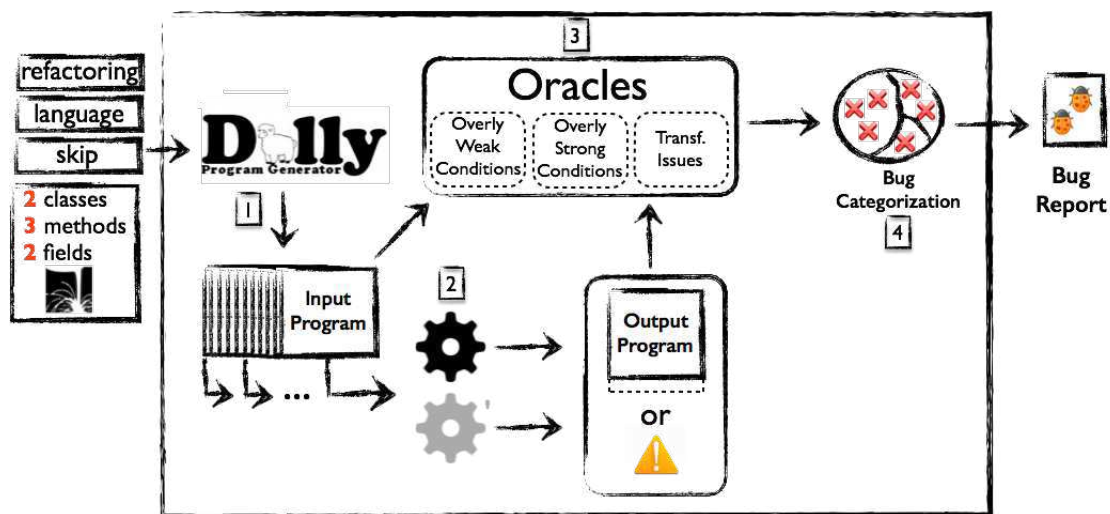


Figure 4.1: A technique for scaling testing of refactoring engines.

## 4.2 DOLLY 2.0

In this section, we explain the modified DOLLY (see Section 2.4.2). We add new features to increase its expressiveness, allow larger scopes, and reduce the cost to test the refactoring implementations. We increase the expressiveness of DOLLY by generating programs

considering more Java constructs, such as abstract classes and methods, and interface (Section 4.2.1). We also add a new feature to skip some Alloy instances to reduce the number of generated programs and consequently the time to test the refactoring implementations (Section 4.2.2).

### 4.2.1 New Java Program Constructs

We add more Java constructs in DOLLY to increase its expressiveness and find more kinds of bugs. The first step is modifying the Java meta-model in Alloy used by DOLLY. We need to take care to avoid state explosion when generating the Alloy instances, even with a small scope. Next we explain how we add the new features related to abstract classes and methods, and interface in the Java meta-model. Also, we explain the new well-formedness rules.

#### Abstract Classes and Methods

To represent an abstract method in the Java meta-model used by DOLLY, we allow creating methods without body. We change the multiplicity of the relation `Method -> Body` from `one` to `lone` (see the following fragment of the specification).

---

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

---

Abstract methods can only belong to abstract classes or interfaces as specified next.

---

```
fact abstract_methods {
  all m: Method, c: Class |
    m in c.methods && isAbstract[m] =>
      (isAbstract[c] || isInterface[c])
}
```

---

When a method does not have a body, DOLLY considers it as an abstract method. The following predicate specifies whether a method is abstract.

---

```
pred isAbstract[m:Method] {
  no m.b
}
```

---

DOLLY considers a class as abstract when it contains at least one abstract method. The following predicate specifies whether a class is abstract.

---

```
pred isAbstract[c: Class] {
  some m: c.methods | isAbstract[m]
```

---

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

### Interface

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

---

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

---

DOLLY recognizes an interface when there is a class implementing it. The following fact specifies this rule.

---

```
fact interface {
  all c: Class | isInterface[c.implement]
```

---

All methods of an interface must be abstract as specified in the next predicate.

---

```
pred isInterface[c: Class] {
  all m: c.methods | isAbstract[m]
```

---

We do not implement a new signature representing interfaces to avoid state explosion of Alloy instances. A new signature without relations may increase the number of instances of a specification to:  $\#instances * 2^n$ , where `instances` is the number of instances without the new signature and `n` is the scope of the added signature. We can guide the scope of the interface by specifying an additional constraint to define that a program must have a specific number of classes in which all methods are abstract.

### Well-Formedness Rules

In this new version of DOLLY we investigate the failures related to the generated programs that do not compile for the purpose of reducing the rate of uncompileable programs. We add some well-formedness rules to reduce the number of uncompileable programs considering all the specified constructs without reducing the expressiveness of DOLLY. For example, an abstract method cannot be called. The following fact specifies that there is no abstract method related to a method invocation.

---

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

---

If a concrete class implements an interface, it also must implement all interfaces' methods. We specify this well-formedness rule in the following fact.

---

```
fact allClassesMustImplementMethodsOfItsInterface {
  all c: Class | (#c.implement = 1 && !isAbstract[c]) =>
    (all m: Method | some m2: Method | (m in c.implement.methods) =>
      (m2.id = m.id && m2.param = m.param &&
        m2.acc = m.acc && m2 in c.methods && !isAbstract[m2]))
}
```

---

There is a similar well-formedness rule specifying that if a concrete class extends an abstract class, it must implement all abstract methods of the abstract class. We specify other well-formedness rules related to abstract classes and methods, and interface in the Java meta-model. We provide them in the dissertation's website.

### 4.2.2 Skipping Programs

By default, DOLLY exhaustively searches for all possible combinations yielded by the run command. Even for a small scope, DOLLY may generate thousands of programs. However, the Alloy Analyzer may generate a number of similar consecutive instances [86]. Inspired on the STG technique [29], we allow developers to guide the program generation by skipping some instances. By skipping some consecutive programs we can reduce the number of failures related to the same bug. For a skip  $n$ , which  $n$  is a positive integer, DOLLY generates one



program from an Alloy instance, and jumps the following  $n-1$  Alloy instances. It follows this approach until the Alloy Analyzer has no more instances to generate. We implement the skip mechanism by modifying the DOLLY's source code to discard the skipped Alloy instances instead of translating each one into a program. Figure 4.2 illustrates our technique to skip test inputs.

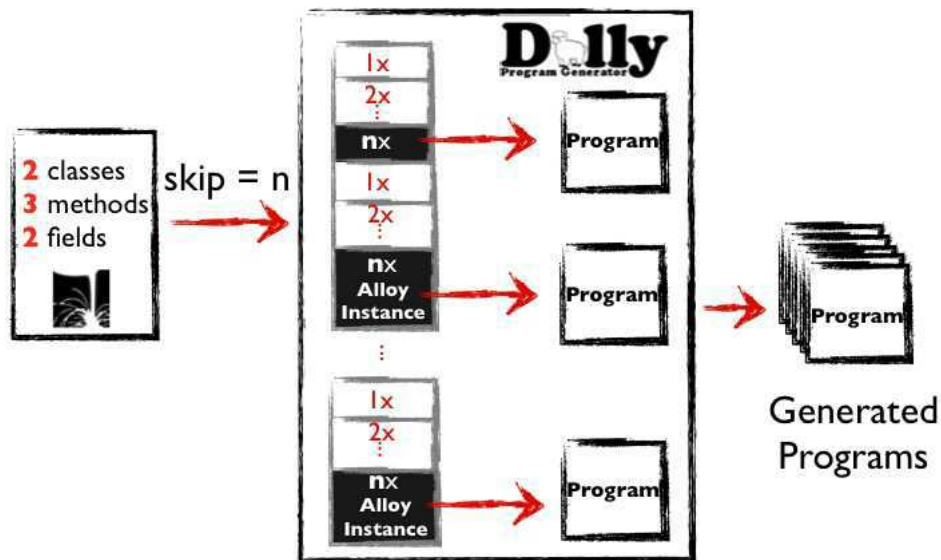


Figure 4.2: Technique used by DOLLY 2.0 to skip programs.

### 4.3 Evaluation

We evaluate our technique to skip programs in 18 Java (Eclipse and JRRT) and C (Eclipse) refactoring implementations with respect to time consumption and bug detection. We present the evaluation of the technique to identify bugs related to overly weak preconditions (compilation errors and behavioral changes) and overly strong preconditions using the Differential Testing technique (DT technique) [80]. We use the oracles explained in Section 2.4.4 to identify the overly weak and overly strong preconditions. All tools and experimental data are available online.<sup>1</sup>

First, we present the experiment definition (Section 6.3.1) and planning (Section 6.3.2). Next, Sections 6.3.3 and 6.3.4 present and discuss the results, respectively. Section 6.3.5 describes some threats to validity and Section 4.3.6 summarizes the main findings.

<sup>1</sup>[http://www.dsc.ufcg.edu.br/~spg/mongiovi\\_thesis.html](http://www.dsc.ufcg.edu.br/~spg/mongiovi_thesis.html)

### 4.3.1 Definition

The goal of this experiment is to analyze our technique to skip programs for the purpose of evaluating it with respect to time consumption and bug detection using skips to generate programs from the point of view of the developers of refactoring engines in the context of Java and C refactoring implementations. In particular, we address the following research questions:

- **Q1** How many bugs, for each refactoring type, the proposed technique can detect bugs?

We measure the number of bugs related to compilation errors, behavioral changes, and overly strong preconditions for each kind of refactoring implementation assessed by our technique.

- **Q2** What is the rate of time reduction and undetected bugs using skips in the technique?

We measure the number of detected bugs and the total time to test the refactoring implementations without skip and using skips of 10 and 25 to generate programs. The total time includes the time to generate the programs, apply the transformations, and execute the automated oracles and bug categorizers.

- **Q3** What is the impact of using skip to generate programs on the time to find the first failure?

We measure the time to find the first failure related to compilation error or behavioral change without skip and using skips of 10 and 25 to generate programs.

### 4.3.2 Planning

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

#### Selection of Subjects

We tested 18 refactoring implementations of Java (Eclipse and JRRT) and C (Eclipse). Eclipse is a widely used refactoring engine in practice and JRRT was proposed to improve the correctness of refactorings by using formal techniques. The evaluated refactorings focus on a representative set of program structures. Moreover, a survey carried out by Murphy et

al. [49] shows the Eclipse JDT refactorings that Java developers use most: Rename, Move Method, Extract Method, Pull Up, and Add Parameter. Three of them are evaluated in this chapter. In a future work we intend to evaluate more kinds of refactorings. We also evaluated all refactoring implementations of Eclipse CDT (C) but one: toggle function. This refactoring needs more than one C file to apply the refactoring, which is not supported by the current version of CDOLLY [47]. Table 4.1 summarizes all evaluated refactorings for Java and C.

### **Instrumentation**

We ran the experiment on a Desktop 3.0 GHz core i5 with 8 GB RAM running Ubuntu 12.04 with JDK 1.6. We tested the refactoring implementations of Eclipse JDT 4.3, Eclipse CDT 8.1, and JRRT (02/03/13). We use SAFEREFACTORIMPACT [46] to evaluate whether a transformation preserves the program behavior and SAFEREFACTOR for C to detect behavioral changes in C refactoring implementations [47]. We use DT technique to detect the overly strong preconditions. The time limit used by SAFEREFACTORIMPACT and SAFEREFACTOR generate tests is 0.3 second. This time limit is enough to test transformations applied to small programs [46]. Finally, we use DOLLY with the skip feature but without the new constructs to generate the programs.

We used the same Alloy specifications proposed before [77] as input parameter of JDOLLY to test the Java refactoring implementations. We did not use the new specification of DOLLY with the new constructs to avoid greatly increasing the number of Alloy Instances. We wanted a baseline to compare the time and bug detection without using skips to generate programs. We used the scope of two packages, three classes, and at most two fields and three methods to JDOLLY generate the programs. We used the scope of two functions, two variables, two defines, and three statements to CDOLLY generate the programs. We defined some additional constraints for guiding DOLLY to generate programs with certain characteristics needed to apply the refactorings. They prevent the generation of programs from which the refactoring under test is not applicable. For example, to test the Pull Up Field refactoring, the program must have at least two classes in a hierarchy, which a subclass contains a field that the super class does not contain. Finally, the oracles save the results in files, which are used by the bug categorization module to categorize the failures in distinct bugs.

### 4.3.3 Results

DOLLY generated 96,129 compilable programs to evaluate all refactorings implementations without skip. We used skips of 10 and 25 to reduce the set of generated programs. DOLLY generated 9,371 and 3,932 compilable programs to those skips, respectively (see Table 4.1).

Table 4.1: Summary of the number of generated programs and the time to evaluate the refactoring implementations.

Refactoring		Generated Programs			Time: Behavioral Change and Compilation Errors (h)						Time: Overly Strong Conditions (h)		
		No skip	Skip 10	Skip 25	Eclipse			JRRT			No skip	Skip 10	Skip 25
					No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25			
Java	Rename Method	8,634	809	361	4.63	0.45	0.18	4.19	0.42	0.15	2.53	0.25	0.09
	Push Down Field	9,117	870	390	4.32	0.42	0.17	1.75	0.17	0.05	0	0	0
	Pull Up Field	8,712	867	345	5.9	0.58	0.24	3.01	0.28	0.10	0	0	0
	Encapsulate Field	1,831	175	68	1.55	0.14	0.05	1.30	0.12	0.06	0.46	0.04	0.01
	Move Method	14,814	1,293	624	8.6	0.72	0.29	6.16	0.63	0.27	5.08	0.58	0.23
C	Rename Global Variable	2,040	198	86	0.61	0.06	0.02	-	-	-	-	-	-
	Rename Local Variable	7,359	786	314	2.15	0.17	0.07	-	-	-	-	-	-
	Rename Define	1,034	101	38	0.23	0.04	0.01	-	-	-	-	-	-
	Rename Function	13,188	1,327	531	3.56	0.22	0.09	-	-	-	-	-	-
	Rename Parameter	5,964	587	233	1.54	0.13	0.06	-	-	-	-	-	-
	Extract Function	7,812	786	314	2.10	0.12	0.05	-	-	-	-	-	-
	Extract Local Variable	7,812	786	314	2.95	0.18	0.07	-	-	-	-	-	-
	Extract Constant	7,812	786	314	2.55	0.17	0.08	-	-	-	-	-	-
<b>Total</b>		<b>96,129</b>	<b>9,371</b>	<b>3,932</b>	<b>37.13</b>	<b>3.40</b>	<b>1.38</b>	<b>16.41</b>	<b>1.62</b>	<b>0.63</b>	<b>8.07</b>	<b>0.87</b>	<b>0.33</b>

Table 4.2: Summary of the detected bugs related to compilation errors.

Refactoring		Eclipse						JRRT					
		Failures			Bugs			Failures			Bugs		
		No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25
Java	Rename Method	595	68	19	1	1	1	0	0	0	0	0	0
	Push Down Field	342	37	15	2	2	2	0	0	0	0	0	0
	Pull Up Field	518	59	26	1	1	1	0	0	0	0	0	0
	Encapsulate Field	238	22	12	1	1	1	0	0	0	0	0	0
	Move Method	214	22	9	2	2	2	3	0	0	1	0	0
C	Rename Global Variable	907	91	33	4	4	4	-	-	-	-	-	-
	Rename Local Variable	3,274	326	129	4	4	4	-	-	-	-	-	-
	Rename Define	391	41	21	4	4	4	-	-	-	-	-	-
	Rename Function	6,165	627	245	5	5	5	-	-	-	-	-	-
	Rename Parameter	2,681	262	108	4	4	4	-	-	-	-	-	-
	Extract Function	5,745	583	231	7	7	7	-	-	-	-	-	-
	Extract Local Variable	3,880	393	157	5	5	5	-	-	-	-	-	-
Extract Constant	4,168	415	158	8	8	8	-	-	-	-	-	-	
<b>Total</b>		<b>29,118</b>	<b>2,946</b>	<b>1,163</b>	<b>48</b>	<b>48</b>	<b>48</b>	<b>3</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>

The proposed technique detected a total of 74 bugs, from which 49 bugs are related to compilation errors (48 in Eclipse and 1 in JRRT), 17 to behavioral changes (14 in Eclipse and 3 in JRRT), and 8 to overly strong preconditions (7 in Eclipse and 1 in JRRT). Using skips of

Table 4.3: Summary of the detected bugs related to behavioral changes.

Refactoring		Eclipse						JRRT					
		Failures			Bugs			Failures			Bugs		
		No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25
Java	Rename Method	0	0	0	0	0	0	286	36	8	1	1	1
	Push Down Field	71	5	4	1	1	1	0	0	0	0	0	0
	Pull Up Field	690	64	29	4	4	4	0	0	0	0	0	0
	Encapsulate Field	0	0	0	0	0	0	0	0	0	0	0	0
	Move Method	2,983	295	126	3	2	2	6,034	592	250	2	2	2
C	Rename Global Variable	107	12	4	1	1	1	-	-	-	-	-	-
	Rename Local Variable	228	48	11	1	1	1	-	-	-	-	-	-
	Rename Define	0	0	0	0	0	0	-	-	-	-	-	-
	Rename Function	0	0	0	0	0	0	-	-	-	-	-	-
	Rename Parameter	133	17	3	1	1	1	-	-	-	-	-	-
	Extract Function	659	71	28	1	1	1	-	-	-	-	-	-
	Extract Local Variable	701	66	41	1	1	1	-	-	-	-	-	-
Extract Constant	597	63	25	1	1	1	-	-	-	-	-	-	
<b>Total</b>		<b>6,169</b>	<b>641</b>	<b>271</b>	<b>14</b>	<b>13</b>	<b>13</b>	<b>6,320</b>	<b>628</b>	<b>258</b>	<b>3</b>	<b>3</b>	<b>3</b>

Table 4.4: Summary of the detected bugs related to overly strong preconditions using DT technique.

Refactoring		Eclipse						JRRT					
		Rejected Behavior Pres. Transf.			Overly Strong Conditions			Rejected Behavior Pres. Transf.			Overly Strong Conditions		
		No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25
Java	Rename Method	4,165	470	151	3	3	3	0	0	0	0	0	0
	Push Down Field	0	0	0	0	0	0	0	0	0	0	0	0
	Pull Up Field	0	0	0	0	0	0	0	0	0	0	0	0
	Encapsulate Field	734	92	24	1	1	1	0	0	0	0	0	0
	Move Method	2,411	320	94	3	3	3	105	9	2	1	1	1
<b>Total</b>		<b>7,310</b>	<b>882</b>	<b>269</b>	<b>7</b>	<b>7</b>	<b>7</b>	<b>105</b>	<b>9</b>	<b>2</b>	<b>1</b>	<b>1</b>	<b>1</b>

10 and 25 the technique missed two bugs. The missed bugs are related to compilation errors, and behavioral changes in the Move Method refactoring. Tables 4.2, 4.3, and 4.4 summarize the results of the detected bugs related to compilation errors, behavioral changes, and overly strong preconditions, respectively.

Since we tested newer versions of Eclipse JDT and JRRT comparing with our previous works [77; 80], we detected some new bugs that our previous works did not detect. But, we also detected some bugs that we have already detected (those bugs have not been fixed yet in the engines). We reported all new bugs found in the bug tracker of Eclipse and we sent them by email to the JRRT developers.

The total time to evaluate all refactoring implementations without skip to generate programs was 61.61hrs. Using skips of 10 and 25, the technique took 5.89hrs (90% of time reduction) and 2.34hrs (96% of time reduction), respectively. By using both skips the technique missed only 2.7% of the bugs. Table 4.1 summarizes the results related to the time to test the refactoring implementations.

We measured the Time To Find the First Failure (TTFF) in the refactoring implementations under test. The technique took in general a few seconds to find the first failure, which can be related to compilation error or behavioral change (see Table 4.5). In some refactorings, such as Push Down Field and Encapsulate Field, it took some minutes to find the first failure without skip. In case of no failure, we show “n/a”. We show “-” if we do not evaluate.

Table 4.5: Summary of the Time to Find the First Failure (TTFF).

Refactoring		Eclipse			JRRT		
		TTFF (min)			TTFF (min)		
		No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25
J a v a	Rename Method	1.00	0.15	0.68	3.54	2.10	1.74
	Push Down Field	6.95	1.03	0.65	n/a	n/a	n/a
	Pull Up Field	0.13	1.05	0.51	n/a	n/a	n/a
	Encapsulate Field	2.60	1.51	1.03	n/a	n/a	n/a
	Move Method	0.17	0.62	0.61	0.25	1.05	0.58
C	Rename Global Var.	0.10	0.05	0.02	-	-	-
	Rename Local Var.	0.02	0.12	0.05	-	-	-
	Rename Define	0.18	0.2	0.01	-	-	-
	Rename Function	0.06	0.01	0.08	-	-	-
	Rename Parameter	0.20	0.05	0.11	-	-	-
	Extract Function	0.09	0.02	0.01	-	-	-
	Extract Local Var.	0.12	0.04	0.02	-	-	-
	Extract Constant	0.03	0.14	0.01	-	-	-

### 4.3.4 Discussion

Now we discuss issues of the detected bugs related to compilation errors, behavioral changes, overly strong preconditions, and the time to test the refactoring implementations.

#### Compilation Errors

A total of 29,118 transformations applied by Eclipse failed due to compilation errors without skip. Those failures were categorized in 48 bugs. We missed no bug in the refactoring implementations of Eclipse using the skips of 10 and 25. We detected the same bugs in the C refactoring implementations using no skip, a skip of 10, and a skip of 25 to generate programs. Almost half of the transformations failed due to compilation errors in those implementations. So, the skip was useful to reduce the time to test them. For example, the Eclipse CDT does not check whether the new names are keywords and, then introduces compilation errors when applying the transformation. JRRT applied only three transformations (Move Method) with compilation errors. The failures were categorized in one bug.

We found a new bug in the Push Down Field refactoring of Eclipse JDT not detected by previous techniques. The resulting program is presented in Listing 4.1. The transformation moved the field `f` from class `B` to class `C`. Method `C.m` calls the field `B.f`, which is inherited from `A`. However, the field `A.f` is not visible from class `B` because its visibility is package and the classes are in different packages. Then, the resulting program does not compile.

Listing 4.1: Uncompilable output program generated by the Push Down Field refactoring of Eclipse JDT 4.3.

---

```
package p1;
public class A {
    int f = 10;
}

package p0;
import p1;
public class B extends A {}

package p1;
import p0;
public class C extends B {
    public int f = 11;
    public long m() {
        return new B().f;
    }
}
```

---

### Behavioral Changes

JRRT applied 6,320 behavioral changes transformations. We categorized them in three distinct bugs. Eclipse JDT applied 3,744 transformations that change the program behavior while Eclipse CDT applied 2,425 ones. We found 14 distinct bugs related to behavioral changes in the refactorings of Eclipse. For example, we found a new bug in the Pull Up Field of Eclipse JDT. The transformation enables a field to hide another field, which changes the program behavior. Listing 4.2 illustrates the original program generated by JDOLLY and Listing 4.3 shows the resulting program after the Pull Up Field refactoring of Eclipse. In the

original program, the method `k` calls field `A.f` yielding 0. After the transformation, it calls `B.f` yielding 1. The transformation enables the field `B.f` hides field `A.f`. We reported the new bugs to Eclipse. Until the writing of this, we have no feedback from them.

Listing 4.2: Before refactoring.

---

```

class A {
    protected int f = 0;
}
class B extends A {
    public int k() {
        return f;
    }
}
class C extends B {
    private int f = 1;
}

```

---

Listing 4.3: After the Pull Up Field Refactoring of Eclipse JDT 4.3.

---

```

class A {
    protected int f = 0;
}
class B extends A {
    private int f = 1;
    public int k() {
        return f;
    }
}
class C extends B {}

```

---

### Overly Strong preconditions

We found 7 bugs related to overly strong preconditions in Eclipse JDT and 1 bug in JRRT. We found two new bugs in the Move Method refactorings of Eclipse and JRRT. Listing 4.4 shows the original program generated by JDOLLY and Listing 4.5 illustrates the resulting program after the Move Method refactoring of Eclipse JDT. The transformation moved the method `m(int)` from class `B` to class `A`. This transformation does not change the program behavior. All methods of the program yield the same value before and after the transformations. JRRT rejected applying this transformation and reported the following warning: `cannot adjust accessibilities`.

### Time

Our technique took 61.61 hours to evaluate all refactoring implementations under test without skips. DOLLY generates a number of similar programs that may increase the time for testing refactoring engines and potentially detect the same kind of bug. We have observed



Listing 4.4: Before refactoring.

---

```

class A {}
class B extends A {
    A f = null;
    long m(int a) {
        return 0;
    }
}
class C extends B {
    long m(int a) {
        return 1;
    }
    long k() {
        return super.m(2);
    }
}

```

---

Listing 4.5: After the Move Method Refactoring of Eclipse JDT 4.3.

---

```

class A {
    long m(int a) {
        return 0;
    }
}
class B extends A {
    A f = null;
}
class C extends B {
    long m(int a) {
        return 1;
    }
    long k() {
        return super.m(2);
    }
}

```

---

that similar programs tend to be generated consecutively by DOLLY. To alleviate this problem we implemented a feature that allows skipping consecutive test inputs.

When using skips, the refactoring engine developer can detect a number of bugs in a few hours. For example, the technique evaluated 18 refactoring implementations and detected 72 bugs in 2.34 hours using a skip of 25 to generate programs. The developer can run the technique again without skipping while fixing the detected bugs in order to find some missed bugs. Moreover, we can reduce even more the idle time of the developer since the technique found the first failure in the refactoring implementations in a few seconds using a skip of 10 or 25. When there are many failures transformations in a refactoring implementation, the TTFF is similar even varying the skip to generate programs. So, the developer can find a bug in a few seconds, fix the bug, run it again to find another bug, and so on. By using this strategy, the bug categorization step is no longer needed since there is only one failure in each execution. Before a new release, the developer can run the technique without skip to improve confidence that the implementation is correct.

### 4.3.5 Threats to Validity

Next we present the threats to validity of our evaluation.

#### Construct Validity

We reported all bugs found by our technique. Developers accepted some of them and marked others as duplicate or new bugs.

#### Internal Validity

We specify some additional constraints in Alloy for guiding the program generation to each kind of refactoring. Those constraints aim to generate programs with certain characteristics needed to apply the refactoring. It also avoids a state explosion of Alloy instances. However, the additional constraints may hide possibly detectable bugs. We categorize the bugs related to compilation errors and overly strong preconditions by splitting the failing tests based on messages from the engine. We classify behavioral changes based on the program's structure. However, we can miss some bugs if the engine reports the same message to different kinds of bugs. Developers can mitigate this threat when they execute the technique a number of times after fixing the bugs.

#### External Validity

We have only considered a subset of Java and C and a small scope to generate programs. Some of the generated programs may be artificial. We cannot assert that all bugs actually happen in practice. Nevertheless, the technique is useful to warn developers about some overly weak and strong preconditions in the refactoring implementations.

### 4.3.6 Answers to the Research Questions

We now answer our research questions.

- **Q1** How many bugs, for each refactoring type, the proposed technique can detect bugs?

The technique detected a total of 74 bugs, from which 49 bugs are related to compilation errors, 17 to behavioral changes, and 8 to overly strong preconditions using

DT. Yet, to apply some kinds of refactorings the program must have some language constructs currently not supported by the program generator.

- **Q2** What is the rate of time reduction and undetected bugs using skips in the technique?

Using skips of 10 and 25, the technique took 5.89hrs (90% of time reduction) and 2.34hrs (96% of time reduction), respectively. Using both skips the technique missed only 2.7% of the bugs.

- **Q3** What is the impact of using skip to generate programs on the time to find the first failure?

The technique reduced on average 47% and 60% (it took a few seconds) the time to find the first failure using skips of 10 and 25, respectively.

# Chapter 5

## Detecting Transformation Issues in Refactoring Engines

Refactoring transformations have been subject to previous researches [64; 55; 14; 68]. Although there is no unique formal definition for each kind of refactoring, there are common characteristics among them that developers of refactoring engines should follow. For example, a transformation needs to move a field from its original class to a direct superclass to follow the Pull Up Field refactoring definition. Here, we focus on detecting transformations applied by refactoring engines that the resulting program compiles and preserves behavior, but do not follow its refactoring definition. Hereafter, we refer to this kind of problem as transformation issues. In this chapter, we present our oracles to detect transformation issues in refactoring engines and the issue categorizers to classify the failures into distinct issues.

Section 5.1 presents some motivating examples. Sections 5.2 and 5.3 explain the oracles and issues categorizers used in the technique, respectively. Finally, we present our experiment to evaluate the technique in Section 5.4.

### 5.1 Motivating Examples

In this section, we present two examples of transformation issues applied by refactoring engines. First, consider a program presented in Listing 5.1. It contains an abstract class `B` implementing an interface `A`. Class `B` declares an abstract method `m`. By applying the Pull Up Method refactoring using Eclipse JDT 4.5, the transformation creates `m` in interface `A`,

but does not remove `m` from class `B` (see Listing 5.2).

Listing 5.1: Original version.	Listing 5.2: Resulting program.
<pre>public interface A { }  abstract class B implements A {     public abstract int m(); }</pre>	<pre>public interface A {     int m(); }  abstract class B implements A {     @Override     public abstract int m(); }</pre>

Figure 5.1: Pulling up method `B.m()` using Eclipse JDT 4.5 does not remove the method from its original class.

According to some refactoring definitions proposed in the literature [64; 55; 14; 68], the Pull Up Method refactoring intends to remove a method from its original class, create it in the super class, and update all method calls. The transformation presented in Figure 5.1 does not move a method to a super class, but it creates a new method in the super class. It is an issue in the Pull Up Method implementation of Eclipse JDT 4.5. We reported this issue to Eclipse and its developers confirmed it.<sup>1</sup>

The second transformation issue is presented in Figure 5.2. The source program contains two packages: `p1` and `p2`. The first one contains two classes `A` and its subclass `B`. Class `A` declares a method `m`. Package `p2` contains class `B` that also extends class `A` (see Listing 5.3). By applying the Push Down Method refactoring in method `A.m` using JRRT (02/03/13), it removes a class from the program. The transformation moves the method to only one of its subclasses (`p2.B`) and removes the other subclass (`p1.B`). Listing 5.4 illustrates the resulting program. Notice that Eclipse JDT 4.5 correctly applies this same transformation. It pushes down `m` to `p1.B` and `p2.B`. Listing 5.5 illustrates the resulting program.

The Push Down refactoring does not intend to remove classes from the program, this is neither the case of an overly strong condition nor an overly weak condition, but it is a transformation issue in the refactoring implementation. There are other similar scenarios that JRRT applies the transformations without removing entities. For example, JRRT would

<sup>1</sup>[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=473653](https://bugs.eclipse.org/bugs/show_bug.cgi?id=473653)

not remove class (p1 . B) if it had a different name.

Listing 5.3: Original version.

---

```

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

```

---

Listing 5.4: Resulting program.

---

```

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

```

---

Figure 5.2: Pushing down method A.m() using JRRT (02/03/13) removes a class from the program.

Listing 5.5: Resulting program after Eclipse JDT 4.5 pushes down method A.m() in the program presented in Listing 5.3.

---

```

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

```

---

As we see, refactoring engines can apply transformations that compile and preserve the

program behavior, but they do not follow its refactoring definitions.

Transformations can be global and change parts of the code that they are not supposed to. When this scenario happens in practice, mainly for large subjects, the user cannot be aware of all transformation changes. Also, refactoring engines may perform additional transformations they should not do according to its refactoring definition, as we can see in Figure 5.2.

Daniel et al. [10] presented a technique for automatically testing refactoring engines using ASTGEN, a Java program generator, and a set of programmatic oracles that can syntactically compare the programs. Among other kinds of bugs, they found 3 transformation issues in a total of 42 refactoring implementations evaluated. In this chapter, we propose automated oracles to find transformation issues in refactoring engines.

## 5.2 Oracles

In this section, we explain the oracles used by our technique to detect transformation issues: Differential Testing (Section 5.2.1) and Structural Change Analysis (Section 5.2.2).

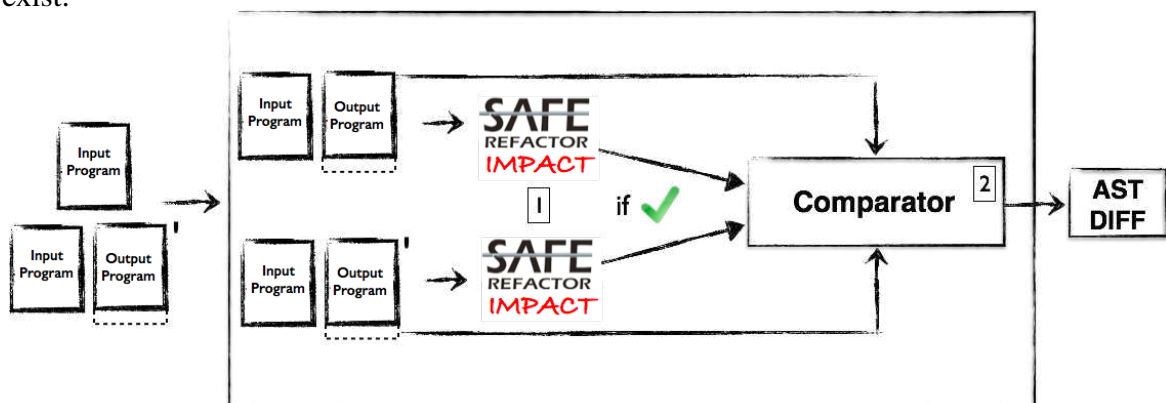
### 5.2.1 Differential Testing Oracle

This oracle receives as input two pairs of programs resulting from transformations applied by refactoring engines for the same input and refactoring type. The same input is provided to both engines. Figure 5.3 illustrates the main steps of this oracle. First, the oracle executes SAFEREFACTORIMPACT to evaluate whether the transformations applied by both engines under test preserve the program behavior (Step 1). The following step is only executed if both transformations compile and preserve the program behavior. In Step 2, the technique compares the outputs generated by the engines. We implement a program to compare two programs concerning their AST (Abstract Syntax Tree). First, it executes a parser and creates the abstract syntactic tree of both programs using the Eclipse JDT API. The comparator checks if the programs contain the same set of packages, classes, interfaces, methods and fields. Next, it compares each pair of classes, methods and fields concerning their modifiers, types (fields and methods), parameters (methods), method bodies (methods), initialization (fields), and imports (classes). It yields the differences between the programs if they exist. If there are differences between the outputs, the oracle reports them and we check if there is a

transformation issue. We manually inspect one pair of programs of each kind of difference to analyze if there is a transformation issue in both or one of the programs. Table 5.1 presents the changes that our comparator considers.

For example, consider that DT oracle receives as input two pairs of programs: the programs presented in Listings 5.3 and 5.4 and the programs presented in Listings 5.3 and 5.5. First, it checks whether both transformations preserve the program behavior using SAFEREFACTORIMPACT. Since SAFEREFACTORIMPACT reports that a transformation preserves the program behavior, the oracle compares their outputs (output 1: Listing 5.4, output 2: Listing 5.5). In this example, the oracle reports the following messages: “*output 2 contains a class ( $p1.B$ ), which the output 1 does not contain*” and “*output 2 contains a method ( $p1.B.m$ ), which the output 1 does not contain.*” We manually analyze these pairs of programs and find that the output 1 is incorrect, since the Push Down refactoring does not intend to remove classes from the program.

Figure 5.3: Differential Testing oracle. In Step 1 the oracle executes SAFEREFACTORIMPACT in both pairs of programs related to transformations of same refactoring type. If the output programs compile and preserve the program behavior, it compares the outputs concerning their AST (Step 2). The oracle reports the differences between the outputs if they exist.



## 5.2.2 Structural Change Analysis Oracle

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

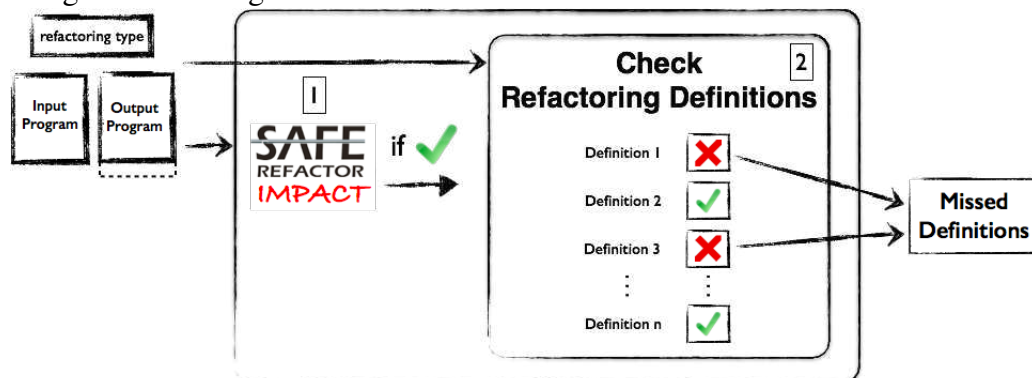


Table 5.1: Changes considered by our program comparator.

Level of change	Kind of change
Program	Set of packages
Package	Set of classes
Class/Interface	Set of Imports
	Set of methods
	Set of fields
	Set of modifiers
Field	Set of modifiers
	Type
	Initialization
Method	Set of modifiers
	Method body
	Return type
	Set of parameters

formations that each refactoring type must or must not perform. For example, for the Pull Up Method refactoring we analyze if a method was removed from its original class and added in the super class. We also analyze additional transformations that must not be performed, such as removing an entity from the program. First, we check if the output program preserves the program behavior using SAFEREFACTORIMPACT (Step 1). If the transformation compiles and preserves the program behavior, we check if it is applied correctly by analyzing the structural changes of the modified program (Step 2). Figure 5.4 illustrates the main steps of this oracle.

Figure 5.4: Structural Change Analysis oracle. In Step 1 the oracle executes SAFEREFACTORIMPACT in the pair of programs related to a refactoring transformation. If the output program compiles and preserves the program behavior, the oracle checks the transformation concerning its refactoring definition.



Next, we explain the definitions used by the technique for four kinds of refactorings. We also define some changes that no transformation must perform. We consider the common characteristics among some refactorings already proposed in the literature [64; 55; 14; 68].

### **Pull Up Method**

**The transformation must:** remove the method from its original class, add the removed method in the direct superclass of its original class, pull up all methods that are in the direct subclasses of the target class (the methods must have the same name, return type, parameters, and body of the refactored method), and update all calls to the refactored methods. **The transformation must not:** move the method when there is another method with the same signature in the target class and apply the transformation when there is no superclass.

### **Push Down Method**

**The transformation must:** remove the method from its original class, add the removed method in the subclass(es) of its original class, and update all calls to the refactored method. **The transformation must not:** move the method when there is another method with the same signature in the target class, and apply the transformation when there is no subclass to push down the method.

### **Pull Up Field**

**The transformation must:** remove the field from its original class, add the removed field to the direct superclass of its original class, pull up all fields that are in the direct subclasses of the target class (the fields must have the same name, type and initializer value of the refactored field), and update all calls to the refactored fields. **The transformation must not:** move the field when there is another field with the same name in the target class, and apply the transformation when there is no superclass.

### **Encapsulate Field**

**The transformation must:** change the field accessibility to private, create a *get* method that returns the field, create a void *set* method that changes the field value, and update all field accesses. The field can only be accessed by the *get* and *set* methods. **The transformation**

**must not:** encapsulate a private field or apply the transformation when there is a *get* or *set* method with the same signature as the methods to be created.

### **For All Refactoring considered by our technique**

Here, we group some transformations that no refactoring considered by our technique can perform. A change is related to a transformation when it follows its refactoring definition.

**The transformation must not:** remove a package, class, method or field of the program that is not related to the transformation, add a package, class, method or field in the program that is not related to the transformation, make a class (without an abstract method) or method abstract, reduce class, method or field accessibility.

For example, consider that SCA oracle receives as input the programs presented in Listings 5.1 and 5.2. First, it checks whether they have the same behavior using SAFEREFAC-TORIMPACT. Since this transformation preserves the program behavior, the oracle checks whether the transformation follows the Pull Up Method refactoring definition described above. It identifies that the *m* method was not removed from its original class *B* and reports the missed definition: "*The m method was not removed from its original class.*"

## **5.3 Issue Categorization**

In this section, we explain how to categorize failures detected by the oracles into distinct issues.

### **5.3.1 Technique using DT oracle**

The output of DT oracle is the set of differences between the outputs of two refactoring engines for the same input and refactoring type. A difference between two programs is a part of the code that one program contains and the other does not contain. We implemented in Java an automated categorizer that clusters the differences. For each kind of difference we define a message. For example, by analyzing the outputs presented in Listings 5.4 and 5.5, DT oracle reports: "*output 2 contains a class ( $p1 . B$ ), which the output 1 does not contain*" and "*output 2 contains a method ( $p1 . B . m$ ), which the output 1 does not contain.*" To classify the failures, we ignore the names of packages, classes, methods, and fields in the messages.

If there is another pair of outputs that contains the same differences, it is flagged with this same message (“*output 2 contains a class, which the output 1 does not contain and output 2 contains a method, which the output 1 does not contain.*”) and we classify them as the same issue. Finally, we manually inspect one pair of programs of each kind of message to analyze if there is a transformation issue in both or one of the programs.

### 5.3.2 Technique using the SCA oracle

The output of SCA oracle for one pair of programs is the set of missed refactoring definitions. As we explained in Section 5.2.2, we use a set of refactoring definitions to analyze each kind of refactoring. We check if the transformation follows all definitions. For example, we categorize the transformation issue presented in Figure 5.1 by the following missed definition: “*The m method was not removed from its original class.*” To classify the failures, we ignore the names of packages, classes, methods, and fields in the messages. When two or more transformations contain the same missed definitions we categorize them as the same issue.

## 5.4 Evaluation

We evaluate our technique using the oracles proposed in this chapter to detect transformation issues in eight refactoring implementations of Eclipse and JRRT. First, we present the experiment definition (Section 5.4.1) and planning (Section 5.4.2). Sections 5.4.3 and 5.4.4 present and discuss the results, respectively. Section 5.4.5 describes some threats to validity and Section 5.4.6 summarizes the main findings.

### 5.4.1 Definition

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

- **Q1** Can the proposed technique using the DT oracle detect transformation issues in the

refactoring engines?

We measure the number of transformation issues detected by the technique using the DT oracle for each kind of refactoring implementation.

- **Q2** Can the proposed technique using the SCA oracle detect transformation issues in the refactoring engines?

We measure the number of transformation issues detected by the technique using the SCA oracle for each kind of refactoring implementation.

- **Q3** Do the DT and SCA oracles detect the same issues?

We analyze the transformation issues detected by both oracles: DT and SCA.

- **Q4** What is the time to test the refactoring implementation using the technique with DT and SCA oracles?

We measure the total time to test each refactoring implementation using the technique with DT and SCA oracles.

### 5.4.2 Planning

In this section, we describe the subjects used in the experiment and its instrumentation. We tested eight refactoring implementations (of four refactoring types) of Eclipse and JRRT.

We ran the experiment on a Desktop computer 3.0 GHz core i5 with 8 GB RAM running Ubuntu 12.04 and JDK 1.6. We tested the refactoring implementations of Eclipse JDT 4.5 and JRRT (02/03/13). We used SAFEREFACITORIMPACT [46] 2.0 with a time limit of 0.5s to generate tests and the change impact analysis parameter activated to evaluate whether a transformation preserves the program behavior. We used a scope of two packages, three classes, up to four methods, and up to two fields to generate the programs for each kind of refactoring. We used DOLLY 2.0 to generate the programs.

### 5.4.3 Results

DOLLY used the Alloy Analyzer to generate up to 1,051,608 instances (Pull Up Field refactoring), which corresponded to 42,064 generated programs using skip of 25. The rate of

compilable generated programs was at least 97% in each refactoring. For some kinds of refactorings, the number of rejected transformations was high. For example, in the Pull Up Field refactoring, both engines did not apply 41,721 transformations, among the set of 42,064 generated programs. On the other hand, JRRT applied all transformations in the Push Down Method refactoring.

The technique using the SCA oracle found 10 and 8 transformation issues in the refactoring implementations of Eclipse and JRRT, respectively. We found no issue in the Pull Up Field refactoring. It took 39.26hrs to evaluate the refactoring implementations of Eclipse and 36.91hrs to evaluate the refactoring implementations of JRRT. Table 5.2 illustrates the result of the technique using the SCA oracle.

The technique using DT oracle found two and three transformation issues in the refactoring implementations of Eclipse and JRRT, respectively. We also found no issue in the Pull Up Field refactoring. It took a total of 75.83hrs to evaluate all refactoring implementations. Table 5.3 illustrates the result of the technique using DT oracle.

Table 5.2: Summary of the evaluation results of Eclipse and JRRT refactoring implementations with our technique using the SCA oracle; Refactoring = kind of refactoring; Scope = scope used by DOLLY to generate programs; P = package; C = class; M = method; F = field; Alloy Instances = number of Alloy instances generated by the Alloy Analyzer; GP (using skip of 25) = number of generated programs using skip of 25 in DOLLY 2.0; CP = compilable generated programs; Transformation Issues = number of different kinds of issues related to incorrect transformations; Time = total time to evaluate the refactoring implementations.

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

#### 5.4.4 Discussion

In this section, we discuss the results of our evaluation concerning the transformation issues detected, issue report, time to test the refactoring implementations, and Dolly 2.0.

Table 5.3: Summary of the evaluation results of Eclipse and JRRT refactoring implementations with our technique using DT oracle; GP (using skip of 25) = number of generated programs using skip of 25 in DOLLY 2.0; App. = applied transformations; BP = behavioral preserving applied transformations; Reject. by = rejected transformations; Different/Equal transf. applied = the outputs of the engines are different/equal; Transf. Issues = number of different kinds of issues related to incorrect transformations.

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

### Transformation Issues Detected by Both Oracles

We detected some transformation issues using this oracle. For example, Figure 5.5 shows an issue in the Pull Up Field refactoring of Eclipse JDT 4.5 detected by the technique using DT oracle. The original program presented in Listing 5.6 contains three classes: A, B that extends A, and C that extends B. Class B declares field `f` initialized by 1 and class C declares field `f` initialized by 2. Applying the Pull Up Field refactoring to move field `f` from class B to class A introduces an issue. In addition to the desired transformation, it removes field `C.f` from the program. Listing 5.7 presents the resulting program.

Listing 5.6: Original version.

```
public class A {}
public class B extends A {
    public int f = 1;
}
public class C extends B {
    public int f = 2;
}
```

Listing 5.7: Resulting program.

```
public class A {
    public int f = 1;
}
public class B extends A {}
public class C extends B {}
```

Figure 5.5: Pulling Up Field B.f using Eclipse JDT 4.5 removes field C.f.

Since the DT oracle can only analyze transformations that both engines apply and the outputs compile and preserve the program behavior, we may miss some issues. Despite

this, the detected issues assisted us to improve the refactoring definitions used to analyze the transformations in the SCA oracle. For example, the transformations presented in Figures 5.2 and 5.5 remove a class and a field from the program, respectively. Based on those detected issues, we added a refactoring definition that no transformation can remove an entity from the program. In addition to reading some proposed informal refactoring definitions, we suggest executing the technique using DT oracle before implementing the SCA oracle for each refactoring type. As we explained before, the detected issues can assist us to define the set of refactoring definitions used by the SCA oracle. We implemented the SCA and DT oracles in Java and executed the experiment using the new version of DOLLY that can generate programs with abstract classes and methods, and interfaces.

The technique using the SCA oracle detected all issues detected by the technique using DT oracle. The technique using the DT oracle did not detect some issues because we only analyze the transformations applied by both engines that compile and preserve the program behavior. Figure 5.6 shows an issue in the Encapsulate Field refactoring of Eclipse JDT 4.5 detected by the technique using the SCA oracle. The original program presented in Listing 5.8 contains class `A`, which declares field `f` and method `getF` returning 0. Applying the Encapsulate Field refactoring in field `f`, choosing the default *get/set* names, the transformation does not create the correct `getF` method because there is a method with the same signature in the class (see Listing 5.9). So, the field is not correctly encapsulated. The engine should ask the user if he would like to choose other *get/set* names or to cancel the transformation.

### Issue Report

We reported all transformation issues detected in Eclipse. So far, they confirmed some issues and rejected or marked others as duplicate. We did not report the issues detected in JRRT since there is no one in charge of it. One of the issues that we reported to Eclipse is related to encapsulating a private field. According to Fowler, only public fields can be encapsulated [14]. We cited Fowler's book to Eclipse developers and they answered that this book is out of date. On the other hand, NetBeans suggests to its developers this book to understand the refactorings.<sup>2</sup> Another issue that we reported to Eclipse was rejected and after our ar-

---

<sup>2</sup><http://wiki.netbeans.org/Refactoring>



Listing 5.8: Original version.	Listing 5.9: Resulting program.
<pre> public class A {     int f;     public int getF() {         return 0;     } } </pre>	<pre> public class A {     private int f;     public int getF() {         return 0;     }     public void setF(int f) {         this.f = f;     } } </pre>

Figure 5.6: Encapsulating field B.f using Eclipse JDT 4.5 does not implement a correct *get* method because there is a method with the same signature.

gumentation they confirmed it. This kind of issue or anomaly introduced by the refactoring engine is still somewhat difficult to confirm by refactoring engine developers, because they implement the refactorings based on their own definitions.

### Time to Test the Refactoring Implementations

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

## Dolly 2.0

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

---

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

---

The average rate of compilable programs in DOLLY 1.0 is 68.8% [77]. We added some constraints related to all constructs of the Java metamodel implemented in DOLLY 2.0 to reduce this rate of uncompileable programs. After adding the new constraints, we have reached a rate of 99.5% of compilable programs generated by DOLLY 2.0. In the Encapsulate Field refactoring 100% of the generated programs compile. The lowest rate is 97.2% in the Pull Up Method refactoring.

Despite the fact that the new constraints have reduced the number of Alloy instances, the Pull Up Field specification has 1,051,608 instances using a scope of three classes and two methods, fields, and packages. This small scope coupled with a high number of Alloy instances indicates the expressiveness of DOLLY 2.0. In the previous technique, DOLLY 1.0 generated at most 30,186 Alloy instances to generate useful programs to find bugs in the refactoring implementations [77]. After the addition of the new constructs, DOLLY 2.0 had to deal with a number of Alloy instances 30 times higher than DOLLY 1.0, which increased the cost to test the refactoring implementations. Furthermore, we have to deal with memory leaks in the Eclipse API. To alleviate these problems and reduce the costs to run the experiment we choose a skip of 25 to generate programs.

### 5.4.5 Threats to Validity

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

#### 5.4.5.1 Construct Validity

Construct validity refers to whether the transformation issues that we have detected are indeed incorrect transformations performed by the refactoring engines. The definition of a transformation issue is strictly related to the refactoring definition used by developers of refactoring engines. They are not only concerned with implementing the pure refactoring. Sometimes, they include features that are not presented in the refactoring definitions. For example, the developers of Eclipse agree on encapsulating a private field while Fowler asserts in his book that only public fields can be encapsulated [14].

#### 5.4.5.2 Internal Validity

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

#### 5.4.5.3 External Validity

We evaluated eight refactoring implementations of Eclipse and JRRT. A survey carried out by Murphy et al. [49] shows that Java developers commonly use Pull Up refactoring. We evaluated Pull Up Field and Pull Up Method refactorings. We plan to evaluate more kinds of refactorings and other refactoring engines, such as NetBeans.

### 5.4.6 Answers to the Research Questions

Next, we answer our research questions.

- **Q1** Can the proposed technique using the DT oracle detect transformation issues in the refactoring engines?

Yes. The technique using DT oracle found one issue related to transformation issue in the Pull Up Method of Eclipse, one in the Push Down Method of Eclipse, one in the Encapsulate Field of JRRT, one in the Pull Up Method of JRRT, and one in the Push Down Method of JRRT.

- **Q2** Can the proposed technique using the SCA oracle detect transformation issues in the refactoring engines?

Yes. The technique using the SCA oracle found three issues related to transformation issue in the Encapsulate Field of Eclipse, five in the Pull Up Method of Eclipse, two in the Push Down Method of Eclipse, two in the Encapsulate Field of JRRT, one in the Pull Up Method of JRRT, and five in the Push Down Method of JRRT.

- **Q3** Do the DT and SCA oracles detect the same issues?

The technique using the SCA oracle detected all issues related to transformation issues detected by the technique using DT oracle. DT technique did not detect three issues in the Encapsulate Field of Eclipse, four in the Pull Up Method of Eclipse, one in the Push Down Method of Eclipse, one in the Encapsulate Field of JRRT, and four in the Push Down Method of JRRT.

- **Q4** What is the time to test the refactoring implementation using the technique with DT and SCA oracles?

The oracles had a similar cost to test the refactoring implementations in this study by using the technique with DT and SCA oracles. The technique using DT oracle took 75.83hrs to evaluate the refactoring implementations of Eclipse and JRRT, while the technique using the SCA oracle took 76.17hrs to evaluate the same implementations.

## Chapter 6

# Detecting Overly Strong Preconditions in Refactoring Engines

Previous work proposes techniques for detecting overly strong preconditions in refactoring implementations. For example, Vakilian and Johnson [87] use refactoring alternate paths to identify usability problems related to overly strong preconditions in refactoring engines. They discover usability problems by analyzing the interactions of users that had problems with tools in general. Our previous work uses Differential Testing [43] to automatically identify transformations rejected by refactoring engines due to overly strong preconditions (DT technique) [80]. To use this technique, developers need to implement a program that automatically applies the refactorings of at least two refactoring engines. However, it can only be used if the engines implement the same refactoring. Additionally, setting up other refactoring engine to automatically apply the transformations may be costly, if the developer is not familiar with the other refactoring engine code.

In this chapter, we present our technique to identify overly strong preconditions in refactoring engines by disabling some preconditions. Section 6.1 presents a motivating example. Next, Section 6.2 describes each step of our technique. Finally, Section 6.3 explains our experiment to evaluate the technique.

### 6.1 Motivating Example

In this section, we present a transformation rejected by Eclipse due to overly strong precon-

dition. Consider part of a program that handles queries to a database, which provides support for two database versions. Each database version is implemented in a class: *QueryV1* (database version 1) and *QueryV2* (database version 2). They make it easy for client code to swap in support for one version, or another. Those classes extend a common abstract class *Query*, which declares an abstract method *createQuery*. This method is implemented in each subclass in a different way. A query created by *createQuery* method is executed by *doQuery* method. Notice that, this method is duplicated in both subclasses: *QueryV1* and *QueryV2*. Listing 6.1 illustrates part of the program.

We can pull up the *doQuery* method to remove duplication. Using Eclipse JDT 2.1 to apply this refactoring, it warns that the *doQuery* method does not have access to *createQuery* method. This precondition checks whether after the transformation the pulled up method still has access to all methods that it calls. However, *createQuery* method already exists as an abstract method in the *Query* class, which indicates that this precondition is overly strong. This bug was reported in Eclipse's bug tracker.<sup>1</sup> Kerievsky reported it when he was working out mechanics for a refactoring to introduce the Factory Method pattern [31]. He argued that “*there should be no warnings as the transformation is harmless and correct.*” The Eclipse developers fixed this bug. Listing 6.2 illustrates a correct resulting program applied by Eclipse JDT 4.5. We found more than 40 bugs related to overly strong preconditions in the bug tracker of Eclipse. As of this writing, the Eclipse developers have already confirmed and fixed more than 50% of them.

We also investigated the test suite of 10 refactorings from Eclipse JDT 4.5 and JRRT: Rename Method, Rename Field, Rename Type, Add Parameter, Encapsulate Field, Move Method, Pull Up Method, Pull Up Field, Push Down Method, and Push Down Field. We classified a total of 2,559 assertions and find that 32% of them are concerned to overly strong preconditions. We consider the following kind of assertion as concerned to overly strong preconditions in the test suite of Eclipse. It checks if Eclipse applies the transformation.

---

```
1 assertTrue("precondition was supposed to pass",
2           !result.hasError())
```

---

In the test suite of JRRT we identified one kind of assertion related to overly strong preconditions as presented next. A test failure indicates that the refactoring implementation may

---

<sup>1</sup>[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=39896](https://bugs.eclipse.org/bugs/show_bug.cgi?id=39896)

Listing 6.1: Pulling Up doQuery method is

rejected by Eclipse JDT 2.1.

---

```

public abstract class Query {
    protected abstract SDQuery createQuery();
}

public class QueryV1 extends Query {
    public void doQuery() {
        SDQuery sd = createQuery();
        //execute query
    }
    protected SDQuery createQuery() {
        //create query for the database version 1
    }
}

public class QueryV2 extends Query {
    public void doQuery() {
        SDQuery sd = createQuery();
        //execute query
    }
    protected SDQuery createQuery() {
        //create query for the database version 2
    }
}

```

---

Listing 6.2: Correct resulting program version.

---

```

public abstract class Query {
    protected abstract SDQuery createQuery();
    public void doQuery() {
        SDQuery sd = createQuery();
        //execute query
    }
}

public class QueryV1 extends Query {
    protected SDQuery createQuery() {
        //create query for the database version 1
    }
}

public class QueryV2 extends Query {
    protected SDQuery createQuery() {
        //create query for the database version 2
    }
}

```

---

have overly strong preconditions.

---

```

1 fail("Refactoring was supposed to succeed;
2   failed with " + rfe)

```

---

This way, we observe that Eclipse and JRRT developers are indeed concerned with identifying overly strong preconditions in their refactoring implementations. Moreover, they may not seem to have a systematic way to create the test cases to test the refactoring implementations with respect to overly strong preconditions. In this chapter, we propose a technique to help developers of refactoring engines to identify them.

## 6.2 Detecting Overly Strong Preconditions by Disabling Preconditions

In this section, we explain the technique we propose to detect overly strong preconditions in refactoring implementations. Section 6.2.1 presents an overview of our technique and Section 6.2.2 describes it by using an example. Next, we explain in more details some steps of it: identification of different kinds of messages reported by the refactoring engine

(Section 6.2.3), and the process of applying transformations to allow disabling the execution of the identified preconditions (Section 6.2.4).

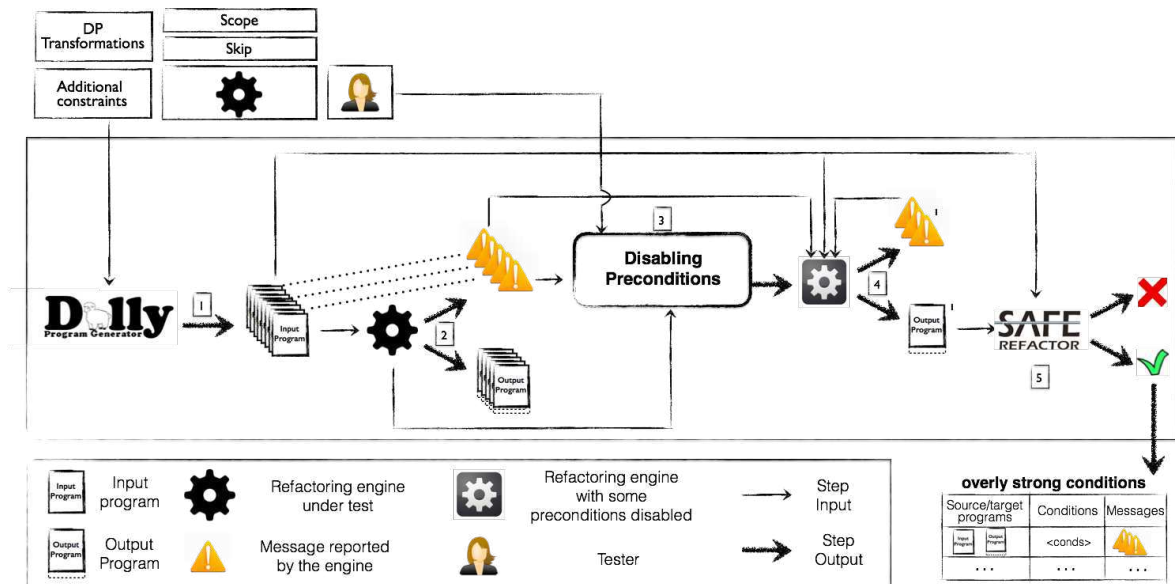


Figure 6.1: A technique to identify overly strong preconditions. First, we generate the programs using JDOLLY (Step 1). For each generated program, we try to apply the transformation (Step 2). Next, for each kind of message reported by the engine, we inspect its code and manually identify the refactoring preconditions that can raise it. We perform transformations in the refactoring engine code to allow disabling the execution of the identified preconditions that prevent the refactoring application (Step 3). Then, for each rejected transformation we try to apply the transformation again using the refactoring engine with the preconditions that raise the reported messages disabled (Step 4). If the transformation preserves the program behavior according to SAFEREFACITORIMPACT, we classify the disabled preconditions as overly strong (Step 5).

### 6.2.1 Overview

Our technique receives as input a refactoring engine, the Disabling Precondition (DP) transformations used to allow disabling the preconditions, and some parameters to JDOLLY, such as skip, scope, and additional constraints. The DP transformations facilitate and systematize the process of changing the code to allow disabling preconditions. Each precondition checks



whether the transformation may introduce a specific problem in the program, which can result in compilation errors or behavioral changes. The technique reports the set of overly strong preconditions identified in the refactoring engine. Figure 6.1 illustrates the main steps of our technique.

First, JDOLLY automatically generates programs as test inputs (Step 1). Next, the refactoring engine under test attempts to apply the transformations to each generated program. If the refactoring engine rejects a transformation, we collect the messages reported to the user (Step 2). For each kind of message, we inspect the refactoring implementation code and manually identify the code fragments related to a precondition that raise it. We assume, for each refactoring type, that there is one precondition related to each kind of message. Then, we change the refactoring engine code, by adding an *if* statement, to allow disabling the execution of the identified precondition using the DP transformations (Step 3).

Once we have already changed the refactoring implementation code to allow automatically disabling the preconditions, we evaluate them. For each refactoring transformation rejected by the refactoring engine, the technique automatically disables in the refactoring engine code the preconditions that raise the reported messages. Next, the refactoring engine automatically tries to apply the same refactoring transformation again with some preconditions disabled (Step 4). If the refactoring engine reports other message, it repeats the process until the engine applies a transformation. If the modified refactoring implementation applies the transformation and the result preserves the program behavior according to SAFEREFAC-TORIMPACT [46], then the technique classifies the set of disabled preconditions as overly strong (Step 5). Otherwise, it analyzes the next rejected refactoring transformation. Once we classify a precondition as overly strong, we do not evaluate it again using other inputs generated by JDOLLY. All steps are automated, except Step 3.

### 6.2.2 Example

In this section, we explain all steps of our technique by using the program presented in Listing 6.3 as input. It contains class *A* and its subclass *B*. Class *A* declares method *k* and class *B* declares methods *k*, *m*, and *test*. Method *B.test* calls method *B.m*, which calls method *B.k* yielding 92. If we try to apply the Pull Up Method refactoring to move *B.m* to class *A* using Eclipse JDT 4.6, it reports the following message: *Method “B.k(…)” referenced in one*

of the moved elements is not accessible from type “A.” The Eclipse developers may prevent the refactoring application because the method to be moved ( $m$ ) calls a method from its origin class ( $B.k$ ). Therefore, when we pull up the method, the transformation could introduce compilation errors if  $B.k$  was not accessible from the destination class  $A$ . However, in this case, all methods are public, and thus they are accessible from any class, which indicates that this precondition may be overly strong. We can apply this transformation without changing the program behavior. Listing 6.4 illustrates a possible correct resulting program. Methods  $B.test$  and  $B.m$  yield value 92 in both versions of the program. In what follows, we explain how we can identify this overly strong precondition using our proposed technique.

Listing 6.3: Pulling Up method  $B.m()$  to class  $A$  is rejected by Eclipse JDT 4.6.      Listing 6.4: A possible correct resulting program version.

---

<pre> <b>public class</b> A {     <b>protected long</b> k(<b>long</b> a){         <b>return</b> 6;     } }  <b>public class</b> B <b>extends</b> A {     <b>public long</b> m(){         <b>return</b> <b>new</b> B().k(2);     }     <b>public long</b> k(<b>int</b> a){         <b>return</b> 92;     }     <b>public long</b> test(){         <b>return</b> m();     } } </pre>	<pre> <b>public class</b> A {     <b>protected long</b> k(<b>long</b> a){         <b>return</b> 6;     }     <b>public long</b> m() {         <b>return</b> <b>new</b> B().k(2);     } }  <b>public class</b> B <b>extends</b> A {     <b>public long</b> k(<b>int</b> a){         <b>return</b> 92;     }     <b>public long</b> test(){         <b>return</b> m();     } } </pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

---

Suppose we want to test the Pull Up Method implementation of Eclipse. First, our technique generates the input programs using JDOLLY. The program presented in Listing 6.3 is one of the programs generated by JDOLLY to test Pull Up Method refactoring implementations. Next, it attempts to apply the transformations, collects the messages reported by the

refactoring engine when it rejects transformations, and inspect the refactoring engine code to identify the refactoring preconditions that raise the messages. Initially, we search in the *refactoring.properties* file for each reported message. In this example, we find the following declaration: This message is represented by the *PullUpRefactoring\_method\_not\_accessible* field from the *RefactoringCoreMessages* class. We search for the methods in the refactoring implementation code that use this field. Only *checkAccessedMethods* method uses this field.

---

```
PullUpRefactoring_method_not_accessible =  
    Method {0} referenced in one of the moved  
    elements is not accessible from type {1}
```

---

Listing 6.5 illustrates part of this method. It contains code fragments of a precondition, which checks if each method called from the moved method is accessible from the destination class. The precondition result is stored in the *isAccessible* variable. If this precondition is not satisfied (Line 4), Eclipse creates the appropriate message (Line 5) and adds an error status in the result of this transformation (Line 6). Before applying the transformation, Eclipse checks if the *result* variable contains some warning or error messages and reports to them.

Listing 6.6 illustrates the same method after we change the code to allow disabling the precondition. It only disables the error status addition (Line 6). By disabling Line 7 (by setting *cond1.isEnabled* to *false*), Eclipse does not report the message and can continue its execution. Notice that, we only need to find a code fragment (inside a method) adding a warning or error status related to the reported message to allow disabling the precondition. When there is more than one method in the refactoring engine code using the field representing the reported message, we need to change the refactoring engine code to allow disabling all of them. For each refactoring type, we change the refactoring engine code to allow disabling the preconditions for all kinds of messages that we collect when it rejects transformations in the programs generated by JDOLLY.

Step 4 automatically disables some preconditions to evaluate whether they are overly strong. For each program generated by JDOLLY that the refactoring engine rejects the transformation, our technique disables the preconditions that prevent the refactoring application. To disable the precondition of our example, our technique automatically sets *isEnabled* field of *ConditionsPullUpMethod.cond1* to *false*. After disabling this precondition, it attempts

Listing 6.5: Original code snippet of Eclipse JDT.

---

```
1 private RefactoringStatus checkAccessedMethods (...) throws
    JavaModelException {
2 final RefactoringStatus result = new RefactoringStatus ();
3 ...
4 if (!isAccessible) {
5     final String msg = Msgs.format(RefactoringCoreMessages.
        PullUpRefactoring_method_not_accessible ,...);
6     result.addError(message, JavaStatusContext.create(method));
7 }
8 ...
9 }
```

---

Listing 6.6: Code snippet after disabling a Pull Up Method refactoring precondition using Transformation 2.

---

```
1 private RefactoringStatus checkAccessedMethods (...) throws
    JavaModelException {
2 final RefactoringStatus result = new RefactoringStatus ();
3 ...
4 if (!isAccessible) {
5     final String msg = Msgs.format(RefactoringCoreMessages.
        PullUpRefactoring_method_not_accessible ,...);
6     if (preconditionsPullUpMethod.cond1.isEnabled())
7         result.addError(message, JavaStatusContext.create(method));
8 }
9 ...
10 }
```

---

to apply the Pull Up Method transformation again. In this example, Eclipse can apply this transformation with this precondition disabled (Listing 6.4).

The last step of our technique consists of analyzing whether the output program compiles and preserves behavior according to SAFEREFACTORIMPACT (Chapter 3). To analyze this transformation, SAFEREFACTORIMPACT receives as input the programs shown in Listings 6.3 and 6.4. First, it identifies the public and common impacted methods [46]. It decomposes a coarse-grained transformation into smaller transformations and for each one, it identifies the set of impacted methods. In this example, we have two small-grained transformations: remove method *B.m* and add method *A.m*. Since there is no other *m* method in the hierarchy, the small-grained transformations only impact these methods. Next, it identifies the methods that directly or indirectly call the impacted methods. In this example, *B.test* calls *B.m* (original program) and *A.m* (modified program). Therefore, *B.m*, *A.m*, and *B.test* are impacted by the transformation. Only these methods may change the behavior after the transformation. SAFEREFACTORIMPACT generates tests only for the impacted methods common to both programs (*B.m* and *B.test*). Finally, it runs the test suite on both program versions and evaluates the results. The test cases pass in both programs and SAFEREFACTORIMPACT reports that the transformation preserves the program behavior.

Since SAFEREFACTORIMPACT classifies this transformation as a refactoring, our technique classifies this precondition as overly strong. In this example, this precondition checks whether the *B.k* method (called from the moved method *m*) is accessible from *A* class. The precondition should be satisfied, but it is not. This same precondition may be needed to avoid some incorrect transformations that introduce compilation errors in the resulting program for other inputs. Developers need to reason about adjusting the precondition for avoiding preventing correct transformations, such as this transformation. Eclipse developers confirmed this bug.<sup>2</sup>

### 6.2.3 Identifying Messages

This section explains how to identify the different kinds of messages reported by the refactoring engine when it rejects refactoring transformations. The first step of our technique consists of generating programs as test inputs using JDOLLY. For each generated program,

---

<sup>2</sup>[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=399788](https://bugs.eclipse.org/bugs/show_bug.cgi?id=399788)

we attempt to apply the transformation using the refactoring engine under test. It may apply some transformations and reject others. We collect all messages reported by the refactoring engine when it rejects transformations. Next, we identify the different kinds of messages, among the set of reported messages. To automate this process, we implement a message classifier using a similar approach proposed by Jagannath et al. [29]. The approach ignores the parts of the message that contain names of packages, classes, methods, and fields. For example, the template of the message reported by Eclipse in Listing 6.3 is: *Method referenced in one of the moved elements is not accessible from type.*

## 6.2.4 Disabling Refactoring Preconditions

In this step, we change the refactoring implementation code to allow disabling the execution of some refactoring preconditions that prevent the engine from applying the refactorings. For example, a Rename Field refactoring implementation must check if there is another field in the same class with the same name. Without this precondition, a transformation can introduce compilation errors in the resulting program. Despite some preconditions avoid incorrect transformations, they may also prevent correct ones because they are too restrictive. We classify this kind of precondition as overly strong precondition. A precondition may be implemented as a set of code fragments of the refactoring implementation, consecutive or not, which can prevent the transformation and report a specific message to the user. A precondition implementation may involve a set of checks in one or more methods.

We disable the code fragment that prevents the refactoring engine to apply a transformation due to an unsatisfied precondition. For each kind of message reported by the refactoring engine, we search for all occurrences of the message in the code to identify the precondition. We automate disabling the preconditions using AspectJ by following the specified transformation templates. Next, we explain how we can manually disable the preconditions using the templates and automatically using aspects.

### 6.2.4.1 Templates

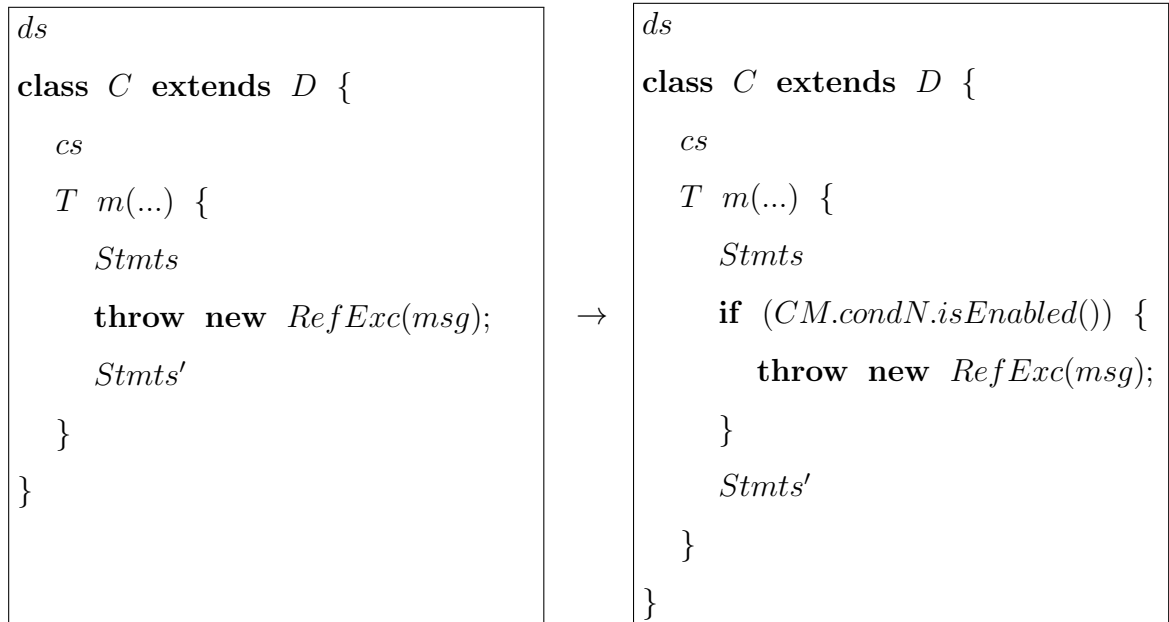
A DP transformation represents a change in the refactoring implementation code that allows disabling the execution of a refactoring precondition. Each DP transformation specifies a Java program template before and after the change. The left hand side template specifies

the method body in a Java program. When the code fragment that we want to disable the precondition, matches the left hand side template, we change the refactoring engine code by following the right hand side template. We specified one DP transformation for JRRT and two for Eclipse. Each DP transformation adds an *if* statement in the refactoring engine code and is applied within a method body. If there is no DP transformation to match, we analyze the minimum changes necessary to allow disabling the code fragments that prevent the refactoring precondition to propose a new kind of DP transformation. If this new kind of DP transformation cannot be used to allow disabling other preconditions, we leave it as a particular case.

The DP transformations contain some Java constructs and meta-variables. The DP transformations from JRRT and Eclipse have the following common meta-variables: *C* specifies a class (it extends a *D* class); *ds* specifies a set of class and interface declarations of the refactoring engine code; *m* specifies a method name; *T* specifies a type name; *Stmts* specifies a sequence of statements; *msg* specifies a message reported to the user by the refactoring engine when it rejects a transformation; and *cs* specifies a set of class structures, such as methods, attributes, inner classes, and static blocks. Meta-variables equal on both sides of a DP transformation means that the transformation does not change them. We create the *ConditionsManagement* class to manipulate the execution of each refactoring precondition (*cond1*, *cond2*, ..., *condN*). For each refactoring type, we create a class (*CM*) that extends *ConditionsManagement*.

JRRT rejects a refactoring transformation when a precondition is not satisfied. As JRRT does not have a graphical user interface, it always throws a *RefactoringException* (*RefExc*) to terminate the execution and report the error message to the user. To disable a refactoring precondition, we apply a transformation for preventing JRRT from throwing the *RefactoringException* when this precondition is unsatisfied. We specify DP Transformation 1 to allow disabling code fragments that prevent preconditions of JRRT.

Eclipse implements a class (*RefactoringStatus*) that stores the outcome of the preconditions checking operation. It contains methods, such as *addError*, *addEntry*, *addWarning*, *createStatus*, *createFatalErrorStatus*, *createErrorStatus*, and *createWarningStatus*. Those methods receive a message and other arguments, describing a particular problem detected during the precondition checking. The methods starting with *create* return a *RefactoringSta-*

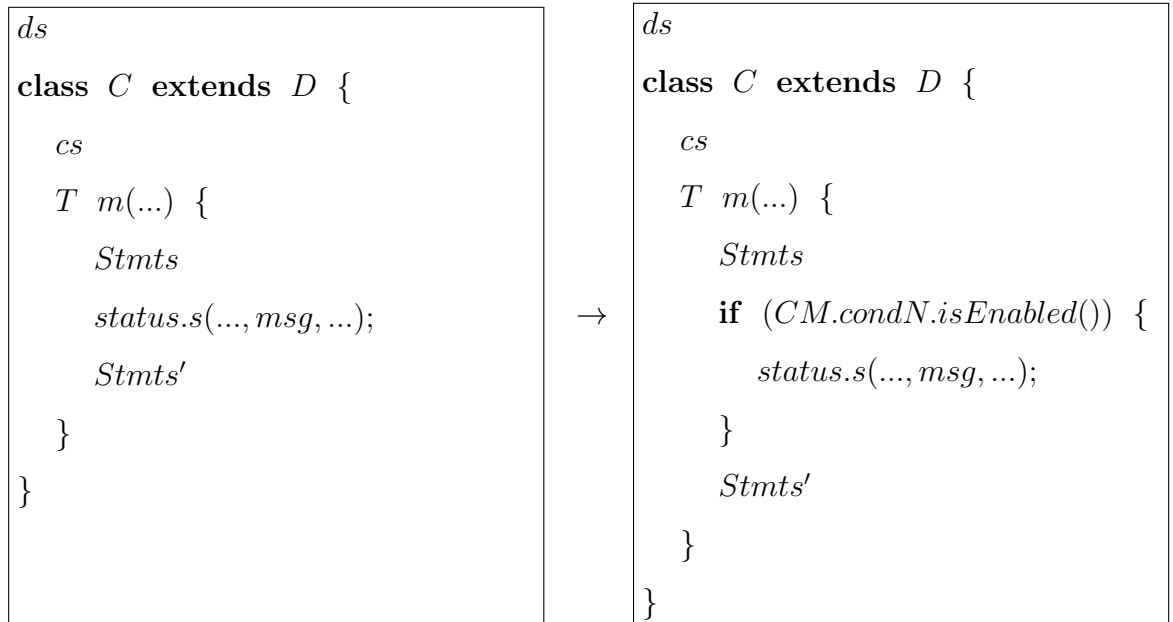
**DP Transformation 1** (Avoid throwing an exception in JRRT)

*tus* object. The messages are stored in the *refactoring.properties* file. They are represented by a field of the *RefactoringCoreMessages* class. They can be directly accessed by a field call or through a variable, parameter of the method, or the return of a method call. The refactoring implementations of Eclipse check the status of a refactoring transformation, in a *RefactoringStatus* object, after evaluating the preconditions. If it contains some warning or error messages, Eclipse rejects the transformation and reports the messages to the user.

We propose the Eclipse DP transformations by analyzing the smallest code fragment, which we need to disable for avoiding the engine to add a new error or warning status in a *RefactoringStatus* object. DP Transformation 2 allows disabling code fragments that prevent preconditions of Eclipse. It has the following specific meta-variables: *status* specifies an object of *RefactoringStatus* type and *s* specifies a method of *RefactoringStatus*.

We use DP Transformation 2 to disable the precondition illustrated in Listing 6.5. *T* matches *RefactoringStatus*, *m* matches *checkAccessedMethods*, *Stmts* matches the sequence of statements from the beginning of the method until Line 5; *status* matches the *result* variable of type *RefactoringStatus*; *m* matches the *addError* method; *WarnMsg* matches *message*; *arg* matches *JavaStatusContext.create(method)*; and *Stmts'* matches the sequence of statements from Line 7 until the end of the method. We use the right hand side of this same DP transformation to change the code to allow disabling the precondition. Listing 6.6 il-



**DP Transformation 2** ⟨Avoid adding a refactoring status in Eclipse⟩

illustrates the modified program. We also implement the transformations specified by the DP transformations using AspectJ. The aspects are available at the article's website.

**6.2.4.2 Aspect-Oriented Implementation**

Aspect-Oriented Programming aims to increase modularity by allowing the separation of crosscutting concerns [33]. Disabling refactoring preconditions can be seen as a crosscutting concern of the refactoring engine. We implemented in AspectJ [32] all DP transformations. The abstract aspect *DisablingPreconditions* (Listing 6.7), declares an abstract pointcut *methodMsg* to collect calls to methods with a *String* parameter (*msg*). It also declares an *around* advice to allow executing only the methods collected in *methodMsg*, which the list *Messages.reportedMsgs* does not contain *msg* (*executePrecond* method). *Messages.reportedMsgs* stores the messages related to the preconditions we want to disable and *msg* is the message related to the evaluated precondition. We implement specific aspects to disable the preconditions of Eclipse and JRRT. They extend *DisablingPreconditions*. Developers can extend the aspects if they need to create more DP transformations. They need to specify the pointcut to collect specific method calls and implement the advice to allow disabling the preconditions.

Listing 6.7: Abstract aspect to disable preconditions.

```
public abstract aspect DisablingPreconditions {
    abstract pointcut methodMsg(String msg);
    void around(String msg): methodMsg(msg) {
        if (executePrecond(msg)) {
            proceed(msg);
        }
    }
    public boolean executePrecond(String msg) {
        return !Messages.reportedMsgs.contains(msg);
    }
}
```

The specific aspect to disable the preconditions of Eclipse avoids adding a new warning or error status in a *RefactoringStatus* object. The *RefactoringStatus* class declares some void methods that add a new status in a *RefactoringStatus* object (methods starting with *add*). It also declares methods that create a new *RefactoringStatus* object, add the status, and return this object (methods starting with *create*). We specify a pointcut and implement an advice for both kinds of methods. The *methodMsg* pointcut collects calls to the *addError*, *addWarning*, and *addEntry* methods of *RefactoringStatus* and the *methodMsgNonVoid* pointcut collects calls to the *createStatus*, *createErrorsStatus*, *createWarningStatus*, and *createFatalErrorStatus* methods. We create this pointcut because those methods return a *RefactoringStatus* object. The refactoring implementations of Eclipse do not add or create a new status when setting the *Messages.reportedMsgs* list with the messages related to the preconditions that we want to disable. Listing 6.8 illustrates the aspect used to disable preconditions of Eclipse. Similarly, we implement the aspect to disable JRRT preconditions.

### 6.2.5 Evaluating Preconditions

In Steps 4 and 5, we evaluate whether the identified preconditions are overly strong. These steps are fully automated. For each rejected transformation, we disable the execution of the preconditions that raise the reported messages. Next, we try to apply the transformation again using the refactoring engine with these preconditions disabled. If it rejects the transformation, we repeat the same process until the refactoring engine applies the transformation (Step 4). When it applies the transformation, we evaluate whether the output compiles and the transformation preserves the program behavior according to *SAFEREFACTORIMPACT*. If *SAFEREFACTORIMPACT* identifies a behavioral change, we conclude that the disabled preconditions are needed to prevent unsafe transformations. Otherwise, we conclude that the

Listing 6.8: Aspect to disable refactoring preconditions in Eclipse.

---

```

public aspect DisablingPreconditionsEclipse extends DisablingPreconditions {
    pointcut methodMsg( String msg):
        call (void RefactoringStatus.addError( String ...)) && args(msg,...) ||
        call (void RefactoringStatus.addWarning( String ...)) && args(msg,...) ||
        call (void RefactoringStatus.addEntry( int , String ...)) && args( int ,msg,...);

    pointcut methodMsgNonVoid( String msg):
        call (RefactoringStatus RefactoringStatus.createErrorStatus( String ...)) && args(msg,...) ||
        call (RefactoringStatus RefactoringStatus.createWarningStatus( String ...)) && args(msg,...) ||
        call (RefactoringStatus RefactoringStatus.createFatalErrorStatus( String ...)) && args(msg,...) ||
        call (RefactoringStatus RefactoringStatus.createStatus( int , String ...)) && args( int ,msg,...);

    RefactoringStatus around( String msg): methodMsgNonVoid(msg) {
        if (executePrecond(msg)) {
            return proceed(msg);
        } else {
            return new RefactoringStatus();
        }
    }
}

```

---

disabled preconditions are overly strong (Step 5). We then proceed to analyze the next input generated by JDOLLY, for which the refactoring implementation rejected the transformation. Once we classify a precondition overly strong, we do not evaluate it again using other inputs generated by JDOLLY, which the refactoring engine rejects the transformation due to it.

## 6.3 Evaluation

### 6.3.1 Research Questions

Our experiment has two goals. The first goal is to analyze the DP technique to detect overly strong preconditions for the purpose of evaluating it with respect to detection of overly strong preconditions and performance from the point of view of the refactoring engine developers in the context of refactoring implementations from Eclipse and JRRT. For this goal, we address the following research questions:

- **Q1** Can the DP technique detect bugs related to overly strong preconditions in the refactoring implementations?

We measure the number of bugs related to overly strong preconditions for each refactoring implementation.

- **Q2** What is the average time to find the first failure using the DP technique?

We measure the time to find the first failure in all refactoring implementations.

- **Q3** What is the rate of overly strong preconditions detected by the DP technique among the set of assessed preconditions?

We measure the rate of preconditions that are overly strong in each refactoring implementation.

The second goal is to analyze two techniques (DP and DT [80]) to detect overly strong preconditions in refactoring implementations for the purpose of comparing them with respect to detection of overly strong preconditions from the point of view of the developers of refactoring engines in the context of refactoring implementations of Eclipse and JRRT. We address the following research question for this goal:

- **Q4** Do DP and DT techniques detect the same bugs?

We measure the bugs detected by both techniques: DP and DT techniques.

## 6.3.2 Planning

In this section, we describe the subjects used in our study and the setup of the experiment.

### 6.3.2.1 Subject selection

We tested 10 refactoring implementations of Eclipse JDT 4.5 and 10 of JRRT (02/03/2013) [68]. Among the evaluated refactorings (Column Refact. of Table 6.1), we evaluated popular refactorings, such as the Rename Method [51; 50] and refactorings that are predominantly performed automatically by developers, such as Encapsulate Field and Rename Class [53].

### 6.3.2.2 Setup

We ran the experiment on two computers with 3.0 GHz Core i5 with 8 GB RAM running Ubuntu 12.04. We used SAFEREFACTORIMPACT [46] 2.0 with a time limit of 0.5 second to generate tests. This time limit is enough to test transformations applied to small programs [46]. We executed the experiment using JDOLLY 1.0 with Alloy Analyzer 4 and

SAT4J solver 2.0.5 to generate the programs with no skip and skips of 10 and 25. To allow disable the preconditions in this experiment, we manually applied the transformations to Eclipse and JRRT code by using the DP transformations.

We used the same Alloy specifications defined before [77; 47] as input parameter of JDOLLY to generate the programs. JDOLLY generates programs with at most two packages, three classes, two fields and three methods to test the refactoring implementations of Eclipse and JRRT. The specification defines some main constraints for guiding JDOLLY to generate programs with certain characteristics needed to apply the refactoring. To test the Rename Class, Method, and Field refactorings, we specified that the programs must declare at least one Class, Method, and Field, respectively. To test the Push Down Method/Field refactorings, the programs must declare a method/field in a superclass. To test the Pull Up Method/Field refactorings, the programs must declare a method/field in a subclass. To test Encapsulate Field and Add Parameter refactorings, the programs must declare at least one public field and method, respectively. Finally, to test the Move Method refactoring, the programs must declare at least two classes. One of the classes must declare a method and a field of the same type of the other class.

The Alloy specification used by JDOLLY also defines additional constraints to minimize the number of generated programs, such as some overloading or overriding methods and some primitive fields. Furthermore, we specified that the programs must have at least one public method for enabling SAFEREFACTORIMPACT to generate tests for evaluating the transformations. We specified a total of nine additional constraints. We automated the refactoring applications of Eclipse and JRRT by investigating the refactoring test suites of them to learn how to apply the refactorings using their source code. We implemented in the same way, only replacing the input programs with the programs generated by JDOLLY. We used the same setup for both evaluated techniques (DP and DT).

### 6.3.3 Summary of the Results

Concerning the JRRT evaluation, we identified 24 refactoring preconditions and found 15 (62%) overly strong preconditions in its refactoring implementations. The DP technique did not detect 3 bugs using a skip of 25 in the Move Method and Push Down Field refactorings of JRRT. It took 0.89h to evaluate all refactoring implementations of JRRT without skip to gen-

erate programs. Using skips of 10 and 25, the technique took 0.28h and 0.09h, respectively. It took on average a minute to find the first failure.

Concerning the Eclipse evaluation, we identified 25 refactoring preconditions and found 15 (60%) different kinds of bugs in its refactoring implementations. The DP technique did not detect 1 bug using skips of 10 and 25 in the Add Parameter refactoring of Eclipse. It took 35.72h to evaluate all refactoring implementations of Eclipse without skip to generate programs. Using skips of 10 and 25, the technique took 4.22h and 1.75h, respectively. It took on average 17.41min to find the first failure using no skip. Using skips of 10 and 25, the technique took on average 2.35min and 1.01min to find the first failure, respectively.

JDOLLY generated 154,040 programs to evaluate all refactorings without skip. Considering all generated programs, the percentage of compilable programs was 72.8%. For future work, we intend to specify more well-formed constraints to minimize uncompileable programs and the cost of analysis. Still, the uncompileable programs do not affect our results concerning the bug detection. Table 6.1 summarizes the evaluation results of JRRT and Eclipse refactoring implementations.

We also compared the proposed technique with the DT technique. The DP technique found nine new bugs that the DT technique cannot find in the refactoring implementations of JRRT and two new bugs in the refactoring implementations of Eclipse. It did not detect five bugs that the DT technique detected in the refactoring implementations of Eclipse. Concerning the use of skips, the DP technique did not detect four bugs using a skip of 25 and one bug using a skip of 10. The DT technique missed no bug using skips of 10 and 25. We calculated the number of missed bugs using skips by comparing with the number of detected bugs using no skip. We need to execute the same technique without skip to find the missed bugs. Table 6.2 summarizes the evaluation results of the comparison between DP and DT techniques.

### 6.3.4 Discussion

In this section, we discuss the results of our evaluation.

Table 6.1: Summary of the DP technique evaluation in the JRRT and Eclipse refactoring implementations; Refact. = Kind of Refactoring; Skip = Skip value used by JDOLLY to reduce the number of generated programs; GP = Number of Generated Programs by JDOLLY; CP = rate of compilable programs (%); N° of assessed preconditions = Number of assessed refactoring preconditions in our study; Overly Strong Preconditions = Number of detected overly strong preconditions in the refactoring implementations; Time (h) = Total time to evaluate the refactoring implementations in hours; Time to First Failure (min) = Time to find the first failure in minutes; "na" = not assessed.

Refact.	Skip	GP	CP (%)	N° of assessed preconditions		Overly Strong Preconditions		Time (h)		Time To First Failure (min)	
				JRRT	Eclipse	JRRT	Eclipse	JRRT	Eclipse	JRRT	Eclipse
Move Method	no	22,905	69	6	3	6	3	0.01	4.50	0.30	10.2
	10	2,290		6	3	6	3	0.01	0.40	0.08	0.60
	25	916		4	3	4	3	0.02	0.19	0.06	1.21
Pull Up Method	no	8,937	72	4	2	2	2	0.12	0.75	0.16	4.92
	10	893		4	2	2	2	0.04	0.10	0.46	5.61
	25	357		4	2	2	2	0.01	0.03	0.18	0.90
Push Down Field	no	11,936	79.1	3	2	1	0	0.10	3.11	0.18	na
	10	1,193		3	2	1	0	0.01	0.30	0.11	na
	25	477		3	2	0	0	0.01	0.12	na	na
Rename Method	no	11,264	79.5	3	3	1	3	0.12	0.06	0.06	0.05
	10	1,126		3	3	1	3	0.01	0.01	0.08	0.06
	25	450		3	3	1	3	0.01	0.06	0.08	0.20
Push Down Method	no	20,544	78.5	3	3	3	1	0.15	7.15	2.53	49.43
	10	2,054		3	3	3	1	0.16	1.09	0.46	4.75
	25	821		3	3	3	1	0.01	0.39	0.20	1.91
Pull Up Field	no	10,928	79.7	1	1	0	1	0.08	0.01	na	0.11
	10	1,092		1	1	0	1	0.01	0.01	na	0.11
	25	437		1	1	0	1	0.01	0.01	na	0.08
Add Parameter	no	30,186	63	4	3	2	2	0.31	11.48	0.36	91.26
	10	3,018		4	3	2	1	0.04	1.61	0.08	9.48
	25	1,207		4	3	2	1	0.02	0.65	0.08	3.66
Encapsulate Field	no	2,000	92.8	0	1	na	1	na	0.01	na	0.33
	10	200		0	1	na	1	na	0.01	na	0.28
	25	80		0	1	na	1	na	0.01	na	0.53
Rename Field	no	19,424	79.2	0	3	na	0	na	5.98	na	na
	10	1,942		0	3	na	0	na	0.44	na	na
	25	776		0	3	na	0	na	0.18	na	na
Rename Type	no	15,916	65.5	0	4	na	2	na	2.67	na	0.11
	10	1,591		0	4	na	2	na	0.26	na	0.05
	25	636		0	4	na	2	na	0.11	na	0.08
Total/Average	no	154,040	72.8	24	25	15	15	0.89	35.72	0.59	17.41
	10	15,399		24	25	15	14	0.28	4.22	0.21	2.35
	25	6,157		22	25	12	14	0.09	1.75	0.12	1.01

Table 6.2: Summary of the comparison between DP and DT techniques using input programs generated by JDOLLY; Refact. = Kind of Refactoring; Skip = Skip value used by JDOLLY to reduce the number of generated programs; DP = DP Technique; DT = DT Technique; Overly Strong Preconditions = Number of detected overly strong preconditions in the refactoring implementations; "na" = not assessed.

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

#### 6.3.4.1 Assessed Preconditions

We identified 24 preconditions of JRRT and 25 preconditions of Eclipse from the messages reported by them when they reject transformations. We relate each reported message to one



precondition for each refactoring type. Table 6.3 illustrates some of the Eclipse and JRRT assessed preconditions considered in our evaluation. For each one, we explain what the precondition checks (fourth column), the message reported by the refactoring engine when the precondition is unsatisfied (fifth column), and if our technique classified it as overly strong in this study (sixth column).

For example, Precondition 1 prevents JRRT to move a method when it overrides (or is overwritten by) different methods before and after the transformation. Without this precondition, the transformation may change the program behavior. However, our technique classified this precondition as overly strong because it also prevents from moving an overwritten method when there is no other method in the program calling it. Precondition 4 avoids the same problem in the Add Parameter refactoring of JRRT, since changing a method signature may change method overriding. Our technique also classified it as overly strong for this refactoring. Preconditions 2 and 3 prevent JRRT to push down or pull up a field to a class that already contains a field with the same name, respectively. Both preconditions avoid introducing compilation errors in the resulting program, since a class cannot declare two fields with the same name. According to this evaluation they are not overly strong.

Precondition 7 prevents Eclipse to move a method to a class that already declares a method with the same name. It avoids introducing compilation errors and behavioral changes in the resulting program. However, our technique found that this precondition is overly strong because the methods can have different types of parameters. Preconditions 8 and 9 prevent Eclipse to rename a method when there is another method in the same package or type in the renamed method hierarchy, with the same name but different parameter types and with the same signature, respectively. They also avoid introducing compilation errors and behavioral changes in the resulting program. For example, it can introduce compilation errors related to reduction of inherited method visibility or can introduce behavioral changes when the renamed method changes the binding of a method call. Our technique classified both preconditions as overly strong because in some cases the renamed method is not public and there is no other method in the program calling it.

Precondition 10 prevents Eclipse to push down a field when there is a method referencing it. It avoids introducing compilation errors when the field does not hide other field, and behavioral changes, otherwise. Precondition 12 prevents Eclipse from adding a parameter

in a method when there is another method in the same class with the same signature. It avoids introducing compilation errors in the resulting program, since a class cannot declare two methods with the same signature. Both preconditions (10 and 12) are not overly strong in this evaluation.

This set of assessed preconditions is a subset of the existing preconditions. The evaluated refactoring implementations may have more overly strong preconditions. Developers may consider programs with different program constructs to detect them. In some cases, preconditions of different refactoring types, such as Preconditions 1 and 4, and Preconditions 2 and 3, are implemented by the same code fragments. The refactoring engine reports the same message when the preconditions are unsatisfied. Even so, we consider them as different preconditions because we analyze each refactoring implementation separately. To test all preconditions of a refactoring implementation, we need to select a set of input programs that leads the refactoring implementation to report all messages when it rejects transformations.

Table 6.3: Subset of Eclipse and JRRT assessed preconditions. Engine = Refactoring engine that contains the precondition; Refactoring = Kind of refactoring; Precondition = precondition checking; Message = reported message when the precondition is unsatisfied; OS (DP) = yes if the DP technique found this precondition as overly strong in this experiment, otherwise no.

Id	Engine	Refactoring	Precondition	Message	OS (DP)
1	JRRT	Move Method	Checks whether the method after the transformation still overrides precisely the same methods as before the transformation	overriding has changed	yes
2		Push Down Field	Checks whether the subclass, where the field will be pushed down, already declares a field with the same name	field of the same name exists	no
3		Pull Up field	Checks whether the target class already declares a field with the same name of the moved field	field of the same name exists	no
4		Add Parameter	Checks whether the method after the transformation still overrides precisely the same methods as before the transformation	overriding has changed	yes
5		Pull Up Method	Checks whether the transformation violates a type constraint	type constraint violated:	yes
6		Push Down Method	Checks whether the transformation can preserve method name bindings. It is unsatisfied when a method name cannot be accessed even with qualifiers	cannot access method <method>	yes
7	Eclipse	Move Method	Checks whether the class, where the method will be moved, already declares a method with the same name	a method with name <method> already exists in the target type	yes
8		Rename Method	Checks whether there is other method, in the same package or type in the renamed method hierarchy, with the same name and number of parameters but different parameter type names of the new method's signature	<package> or a type in its hierarchy defines a method <method> with the same number of parameters, but different parameter type names	yes
9		Rename Method	Checks whether there is other method, in the same package or type in the renamed method hierarchy, with the same new method's signature	<package> or a type in its hierarchy defines a method <method> with the same number of parameters and the same parameter type names	yes
10		Push Down Field	Checks whether some method references the pushed down field	pushed down member <method> is referenced by <method>	no
11		Pull Up Method	Checks whether the methods called by the pulled up method are accessible from the class where the method will be pulled up	method <method> referenced in one of the moved elements is not accessible from type <type>	yes
12		Add Parameter	Checks whether the class, which contains the method to be changed, already declares a method with the same signature of the new method signature	duplicate method in type <type>	no

### 6.3.4.2 Disabling the Assessed Preconditions

We consider some transformations as particular cases because the developers do not follow a pattern to reject a refactoring transformation due to an unsatisfied precondition. We applied

58 DP transformations (22 in JRRT and 36 in Eclipse) and 25 particular cases, which we cannot apply the proposed DP transformations, to allow disabling the execution of the Eclipse and JRRT assessed preconditions in this study. In some places of the code we may apply more than one transformation, since some preconditions of different refactoring types report the same message. Developers can restructure the refactoring engine code to enable using the DT transformations to disable the code fragments of preconditions that we classified as particular cases.

For each precondition, we may apply more than one DP transformation or the same DP transformation more than once because each message may appear in a number of places in the code. The number of messages may impact the performance of Steps 3 and 4 of our technique because we need, for each message, to identify the precondition that raises it and change the refactoring engine code to allow disabling the precondition.

Since JRRT does not have graphical user interface, it throws an exception with the message in 100% of cases and aborts its execution. Eclipse opens a dialog to report the message describing the problem to the user. The user can cancel the refactoring application or continue in some cases. In our study, 36%, 46%, and 14% of the changes we made in the Eclipse code prevent warning, error, and fatal error problems, respectively. Only one change (6%) we cannot assert by static analysis whether it prevents a warning or error message.

Listings 6.9 and 6.10 illustrate the original and modified code of JRRT to allow disabling the execution of a Move Method refactoring precondition (Precondition 1 of Table 6.3), respectively. We can use the DP Transformation 1 to disable this precondition. The transformation was applied to the *unlockOverriding* method from the *AST.MethodDecl* class. This precondition evaluates if a set of overridden methods in the original program (*old\_overridden.equals*) is equal to the set of overridden methods in the program after the transformation (*overriddenMethods*). JRRT rejects the transformation by throwing a *RefactoringException* if the precondition is not satisfied. Our technique classified this precondition as overly strong. We reported this overly strong precondition to the JRRT developers and they classified it as bug due to imprecise analysis. So far, the bug has not been fixed yet.

Among the 25 transformations applied without any DP transformation, there are 6 different kinds of transformations. For example, in some cases we included in the *then* clause of the *if* statement, which disables the precondition, some statements to avoid crashing the

Listing 6.9: Original code snippet of JRRT.

---

```
1 public void unlockOverriding () {
2     ...
3     if (!old_overridden.equals(overriddenMethods()))
4         throw new RefactoringException("overriding has changed");
5     ...
6 }
```

---

Listing 6.10: Code snippet after we change the engine code by using Transformation 1, to allow disabling the execution of a Move Method refactoring precondition.

---

```
1 public void unlockOverriding () {
2     ...
3     if (!old_overridden.equals(overriddenMethods()))
4         if (preconditionsMoveMethod.cond1.isEnabled)
5             throw new RefactoringException("overriding has changed");
6     ...
7 }
```

---

refactoring engine. We also had to add in some cases a *return null* command after the *if* statement, which disables the precondition, to avoid compilation errors in the refactoring implementation code.

In some cases, when we disabled a precondition, the refactoring engine reported other message and we needed to disable other precondition that raises the other reported message, and so on. For seven refactoring implementations, we detected a number of overly strong preconditions at the same time: Move Method, Push Down Method, and Add Parameter of JRRT and Move Method, Pull Up Method, Rename Method, and Rename Type of Eclipse. In the Move Method refactoring of JRRT we needed to disable up to four preconditions at the same time to find a bug (among the set of six assessed preconditions). In the other refactoring implementations, we disabled up to two preconditions.

Regarding Step 3, the first author took around one day of work to understand how Eclipse and JRRT check their refactoring preconditions, raise messages, and reject transformations. After that, she took some minutes to manually change the refactoring engine code to allow disabling each refactoring precondition using the proposed templates. Concerning the particular cases, she also took some minutes.

### 6.3.4.3 Bugs Detected by DP Technique

Among the 49 assessed preconditions, we identified 30 overly strong preconditions (61%) in the Eclipse and JRRT refactoring implementations using the DP technique. For example, Listing 6.3 illustrates a Pull Up Method refactoring rejected by Eclipse due to overly strong precondition (Precondition 11 of Table 6.3). Most of the overly strong preconditions of Eclipse and JRRT found by our technique are related to method accessibilities and name conflicts. Others are related to changes in overriding methods, type constraints violations, shadow declarations, unimplemented features, transformation issues, and changes in method invocations.

JRRT applied transformations to all programs generated by JDOLLY in three refactoring implementations: Encapsulate Field, Rename Field, and Rename Type. We did not detect overly strong preconditions in those refactoring implementations. Different from JRRT, Eclipse rejected some transformations in these refactoring implementations and we found some overly strong preconditions. In both of the refactoring engines we identified 15 overly

strong preconditions using the DP technique.

We reported all detected bugs to the Eclipse developers. So far, they confirmed 47% of them (seven bugs), and did not answer 27% (four bugs). The remaining four bugs were considered duplicate (13%) or invalid (13%). We investigated the duplicated bugs (IDs 434881 and 399183) and reopened them because we think they are not duplicated as the reported messages are different. We also reopened a bug considered invalid by the developers (ID 399181). Developers argued that the refactoring engine does not show a message in this case. However, we tested the same bug in the newest version of Eclipse JDT 4.6 and it still reports the message. So far, they have not answered us. The other invalid bug was in the Pull Up Field refactoring (ID 462994). Developers marked it as invalid because the transformation changes the value of a field not called by any method in the original program. The equivalence notion we adopted does not consider that this kind of change modifies the program behavior. SAFEREFACTORIMPACT only evaluates the behavior of the common public impacted methods. Developers did not fix all confirmed bugs because they have very limited resources who are active committers on the refactoring module. We reported the new bugs of JRRT, not detected by previous technique, to its developers and they consider most of them as bugs due to imprecise analysis or unimplemented features. So far, they left unanswered two of them.

The goal of our technique is to propose a systematic way to evaluate the implemented preconditions. We do not suggest removing the overly strong preconditions found by our technique. By removing them, the refactoring engine may apply incorrect transformations. The developers need to reason about the preconditions and choose the best strategy to slightly weak them without making them overly weak. They can use the DP and DT techniques and our previous technique [77] to detect overly weak preconditions to reason about their preconditions.

#### 6.3.4.4 Time

We computed the time for the automated steps of the DP technique. The time to evaluate the refactoring implementations of JRRT was smaller than the time to evaluate the Eclipse ones in all cases but two: Rename Method and Pull Up Field refactorings. In those refactoring implementations all assessed preconditions of Eclipse are overly strong while this is not true

for JRRT. The execution of the technique finishes when we find that all preconditions under test are overly strong. The execution to evaluate Eclipse finished earlier than the JRRT ones in the Rename Method and Pull Up Field refactorings. However, the total time to evaluate all refactoring implementations of JRRT and Eclipse was 0.89h and 35.72h, respectively.

Our previous study [47] showed that using skips the technique can substantially reduce the time to test the refactoring implementations while missing a few bugs related to overly weak and overly strong preconditions using the DT technique. In this study, we evaluated the influence of skip in the time reduction and bug detection of the DP technique. Using skips of 10 and 25, the total time to evaluate all refactoring implementation reduced in 87% and 94%, while missing 13% and 3% of the bugs, respectively. The total time to evaluate the Move Method refactoring of JRRT and Rename Method refactoring of Eclipse using a skip of 25 was higher than using a skip of 10. In those cases, the technique found that all preconditions under test are overly strong using a skip of 10 earlier than using a skip of 25.

Eclipse took 11.48h and 7.15h to evaluate the Add Parameter and Push Down Method refactorings, respectively. These times were higher than the time to evaluate the other refactoring implementations. JDOLLY generated more programs to evaluate these refactoring types (30,186 for Add Parameter and 20,544 for Push Down Method) and only some of the assessed preconditions of them were classified as overly strong.

The average time to find the first failure in the refactoring implementations of JRRT (few seconds) was also smaller than in the Eclipse ones (17.41min). The average time to find the first failure in Eclipse was affected by some values much higher than the average time, such as the time to first failure in the Push Down Method and Add Parameter refactorings. In the Push Down Method refactoring, JRRT and Eclipse found the first failure after generating 255 and 2,898 programs, respectively. In the Add Parameter refactoring, JRRT and Eclipse found the first failure after generating 328 and 8,258 programs, respectively. The average time to first failure of Eclipse without considering these two higher values is 2.62min. Using skips of 10 and 25, the average time to find the first failure in all refactoring implementations reduced in 85% and 93%, respectively.

Using skips, the developers can run the technique and find a bug in a few seconds or minutes, fix the bug, run the technique again to find another bug, and so on. Or, developers can run the technique to find a number of bugs in a few minutes or hours. Before a release,

they may run the technique without skipping instances to find some missed bugs and improve confidence that the implementations are correct.

#### 6.3.4.5 Comparison of DP and DT techniques using input programs generated by JDOLLY

The techniques are complementary in terms of bug detection. The DP technique detected 11 new bugs (37% of the bugs) that DT technique cannot detect in the Pull Up Field and Add Parameter refactorings of Eclipse and in the Move Method, Rename Method, Push Down Method, and Push Down Field refactorings of JRRT. The DT technique cannot detect some bugs when the other refactoring engine used in the differential testing has overly weak preconditions or also has overly strong preconditions. In the former case, the other refactoring engine applies a transformation that does not preserve the program behavior or the resulting program does not compile. In the latter case, the other refactoring engine also rejects to apply the transformation.

For example, Listing 6.11 presents a program generated by JDOLLY. It contains class *A* and its subclasses *B* and *C*. Classes *A* and *B* contain the field *f* and class *B* declares method *test* that calls field *B.f*, yielding value 1. By using JRRT to apply the Push Down Field refactoring to move *A.f* to class *C*, it rejects this transformation due to an overly strong precondition. By disabling the precondition that prevents the refactoring application, we can apply the transformation without changing the program behavior. Listing 6.12 illustrates the resulting program. Method *B.test* yields value 1 before and after the refactoring. We only detected this overly strong precondition using the DP technique. The DT technique cannot detect it because Eclipse also rejects this transformation. We reported this bug to the JRRT developers and they agreed that this transformation should be applied.

DT technique detected five bugs that DP technique cannot detect in the Push Down Method and Rename Field refactorings of Eclipse. The DP technique cannot detect those bugs because when we disable the code fragments of a precondition, JRRT applies a transformation that includes a cast (two bugs in the Rename Field) or a super modifier (one bug in the Rename Field) in a field call to preserve the program behavior.

For example, Listing 6.13 presents an input program generated by JDOLLY. It contains class *B*, and its subclass *C*. Class *B* contains the field *f1*. Class *C* contains the field *f0* and



declares method *test* that calls *f1* yielding 0. By using Eclipse to rename field *C.f0* to *f1*, it rejects this transformation due to an overly strong precondition. JRRT applies this transformation without changing the program behavior. Listing 6.14 illustrates a resulting program applied by JRRT. Method *C.test* yields 0 before and after the refactoring. We only detected this overly strong precondition using the DT technique. The DP technique cannot detect it because when we disable the precondition, Eclipse applies a non-behavior preserving transformation. It does not include a cast of class *B* in the field call inside method *test*. Without this cast, method *test* calls *C.f1* instead of *B.f1* yielding 1.

Listing 6.11: Pushing down field *A.f* to class

*C* is rejected by JRRT. Bug detected by DP technique and not detected by DT technique because Eclipse also rejects to apply the transformation.

---

```

public class A {
    private int f = 0;
}

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

public class C extends A {}

```

---

Listing 6.12: A possible correct resulting program version applied by JRRT.

---

```

public class A {}

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

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

```

---

The DP technique has some advantages. It reports the set of overly strong preconditions, which may facilitate the weakening of the preconditions and it does not need another refactoring engine. The effort to set up the DP technique consists of the manual step of applying the transformations to disable the preconditions. We modified 1-3 LOC per transformation. Although in the DP technique we need to manually identify the preconditions from the set of reported messages, we propose a systematic strategy to perform this activity by using the DP transformations.

Listing 6.13: Renaming C.f0 to f1 is rejected by Eclipse JDT 4.5. Bug detected by DT technique and not detected by DP technique.

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

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

Listing 6.14: A possible correct resulting program version applied by JRRT.

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

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

An advantage of the DT technique is that it can show useful transformations performed by other refactoring engine (see example in Listing 6.14), which can help developers to identify and fix the overly strong preconditions. However, it needs at least two refactoring engines. When it is possible, developers can run the DP technique and after fixing the detected bugs, they run the DT technique to find more bugs.

The DT technique took on average 17h and 66.2h to test the refactoring implementations of JRRT and Eclipse, respectively. The DP technique took on average 0.89h and 35.7h to test the same refactoring implementations of JRRT and Eclipse. DT technique takes some time to apply the transformations using two refactoring engines.

#### 6.3.4.6 Comparison of DP and DT techniques using input programs of Eclipse and JRRT refactoring test suites

We evaluated the DP technique by replacing the programs generated by JDOLLY with input programs used by developers in the test suites of Eclipse and JRRT. The goal was to analyze if our technique can find overly strong preconditions using other input programs in refactoring implementations of Eclipse and JRRT already evaluated in the previous study (see Section 6.3.2.1). We selected only the input programs used in the test cases that the refactoring engine rejects applying the transformation. We identified 272 input programs to

evaluate the refactoring implementations. The engines reported a total of 71 messages when we attempted to apply the transformations in the evaluated refactoring implementations of Eclipse and JRRT. We related the messages to 71 refactoring preconditions.

The DP technique detected 18 overly strong preconditions not detected by the Eclipse and JRRT developers. The DP technique detected at least one overly strong precondition in 70% and 20% of the evaluated refactoring implementations of Eclipse and JRRT, respectively. We also evaluated the same refactoring implementations using the DT technique and the same input programs. The DT technique detected 15 overly strong preconditions not detected by the Eclipse and JRRT developers. The DT technique detected at least one overly strong precondition in 60% and 10% of the evaluated refactoring implementations of Eclipse and JRRT, respectively.

The DP technique detected eight overly strong preconditions not detected by DT technique in the Pull Up Method, Pull Up Field, Add Parameter, Rename Method, and Encapsulate Field refactorings of Eclipse and in the Push Down Method and Pull Up Method refactorings of JRRT. DT technique detected five overly strong preconditions not detected by DP technique in the Move Method, Pull Up Method, Push Down Method, Rename Field, and Rename Method refactorings of Eclipse. In total, we assessed 71 preconditions and detected 23 overly strong preconditions not detected by the developers.

We cannot detect 17 of the bugs using the current version of JDOLLY. We need to add more Java constructs in JDOLLY to detect them. Besides the 23 detected bugs, we found 12 false-positives in this study. In these bugs, the input programs used by the Eclipse and JRRT test suites do not have public methods. SAFEREFACTORIMPACT did not identify any public method to generate tests and classified the transformations as behavior preserving. We did not have this problem with the input programs generated by JDOLLY, as they have at least one public method. We reported all new bugs to the Eclipse developers but until now they left unanswered. Table 6.4 illustrates the main results of this evaluation.

The developers did not find those overly strong preconditions because they do not seem to have a good support to reason about their preconditions and a systematic strategy to evaluate whether a precondition is overly strong. Furthermore, as they expect the refactoring engine to reject those transformations, they believe that the transformations may change the program behavior. In fact, developers may not have an automated oracle to check behavior

preservation, such as SAFEREFACTORIMPACT.

We also evaluated our technique to detect overly weak preconditions by replacing the programs generated by JDOLLY with input programs used by developers in the test suites of Eclipse and JRRT. We selected only the input programs used in the test cases concerned to overly strong preconditions. In this kind of test case, it is expected that the engine applies a correct transformation. We detected six bugs of compilation errors in the refactoring implementations of Eclipse. The bugs are related to reduction of inherited method visibility, wrong use of generics, and use of undeclared fields and methods. We detected one bug related to behavioral change in the Move Method refactoring of Eclipse and one in the same refactoring of JRRT. In both bugs, the resulting program of the test case throws a *NullPointerException*. Table 6.5 illustrates the main results of this evaluation. Developers did not find these bugs because they may not check whether the expected output of each test case compiles and they may not have an automated oracle to check for behavior preservation, such as SAFEREFACTORIMPACT.

Table 6.4: Summary of the comparison between DP and DT techniques using input programs of Eclipse and JRRT refactoring test suite; Refactoring = Kind of Refactoring; Input programs = Number of selected input programs of the JRRT and Eclipse refactoring test suite; N° of assessed preconditions = Number of assessed refactoring preconditions in our study; Overly Strong Preconditions = Number of detected overly strong preconditions in the refactoring implementations; DP = DP Technique; DT = DT Technique.

Refactoring	Input programs		N° of assessed preconditions		Overly Strong Preconditions			
	JRRT	Eclipse	JRRT	Eclipse	JRRT		Eclipse	
					DP	DT	DP	DT
Move Method	12	27	10	8	0	0	3	4
Pull Up Method	30	25	8	9	1	0	1	1
Push Down Field	0	3	0	3	0	0	0	0
Rename Method	10	92	4	6	0	0	4	3
Push Down Method	12	5	5	3	3	2	0	1
Pull Up field	0	3	0	2	0	0	2	1
Add Parameter	0	5	0	2	0	0	1	0
Encapsulate Field	0	2	0	2	0	0	1	0
Rename Field	2	26	1	4	0	0	2	3
Rename Type	18	0	4	0	0	0	0	0
Total	84	188	32	39	4	2	14	13

Table 6.5: Summary of the evaluation results of our technique to detect overly weak preconditions using input programs of Eclipse and JRRT refactoring test suite; Refactoring = Kind of Refactoring; Input programs = Number of selected input programs of the JRRT and Eclipse refactoring test suite; Compilation Errors = Number of detected bugs related to compilation errors in the refactoring implementations; Behavioral Changes = Number of detected bugs related to behavioral changes in the refactoring implementations.

Refactoring	Input Programs		Bugs related to Compilation Errors		Bugs related to Behavioral Changes	
	JRRT	Eclipse	JRRT	Eclipse	JRRT	Eclipse
Move Method	25	68	0	0	1	1
Pull Up Method	29	64	0	2	0	0
Pull Up Field	0	3	0	0	0	0
Push Down Method	33	50	0	0	0	0
Push Down Field	0	8	0	0	0	0
Encapsulate Field	0	41	0	0	0	0
Add Parameter	2	6	0	0	0	0
Rename Field	42	51	0	2	0	0
Rename Method	29	135	0	1	0	0
Rename Type	88	110	0	1	0	0
<b>Total</b>	<b>248</b>	<b>536</b>	<b>0</b>	<b>6</b>	<b>1</b>	<b>1</b>

### 6.3.5 Threats to Validity

In this section, we discuss some threats to the validity of our evaluation.

#### 6.3.5.1 Construct Validity

Construct validity refers to whether the overly strong preconditions that we have detected are indeed overly strong preconditions. Eclipse considered two bugs reported by us as invalid. Some preconditions that we found may not be overly strong with respect to the equivalence notion adopted by the developers. Different from them, our equivalence notion is related to the behavior of the public methods with unchanged signatures. These methods can exercise methods with changed signatures. Otherwise, the methods with changed signatures may not affect the overall system behavior. So far, they confirmed 47% of the reported bugs.

We have no prior knowledge over the refactoring engines code, since we are not their developers. We may not identify all code fragments related to the preconditions under test. Developers may identify a different set of preconditions and may have better results when

using our technique. Additionally, changing the refactoring engine code may introduce problems in the refactoring engine. It may apply incorrect transformations that do not follow the refactoring definition. We minimize this threat by systematizing the process of disabling the preconditions. We propose DP transformations that each one alters one line of code. Even the special cases change a few lines of code.

Finally, we specify in Table 6.3 some preconditions based on the available source code and documentation of JRRT and Eclipse [16; ?; 72; 73; 71; 74; 68; 67]. Still, some definitions may be incomplete or incorrect as we are not developers of the refactoring engines.

### 6.3.5.2 Internal Validity

A false-positive result of SAFEREFACTORIMPACT indicates that it did not detect a behavioral change. In our technique, a false-positive may incorrectly classify a precondition as overly strong. However, in this study, we manually analyzed each overly strong precondition before reporting it. We only found some false-positives in the experiment using the input programs of the refactoring engines' test suites. The false-positives were related to changes in the standard output or changes in non-public methods that cannot be detected by SAFEREFACTORIMPACT 2.0.

Additional constraints in JDOLLY may hide possibly detectable overly strong preconditions. These constraints can be too restrictive with respect to the programs that can be generated by JDOLLY, which shows that one must be cautious when specifying constraints for JDOLLY. Our current setup for testing Eclipse may have memory leaks. This may have an impact in the time to test its refactoring implementations. Another threat is related to the bugs detected only by DP technique. The DT technique did not identify some bugs because the other engine (JRRT or Eclipse) used to perform differential testing also has overly strong preconditions or overly weak preconditions that allow applying incorrect transformations. Using another refactoring engine to perform differential testing may identify some of those bugs.

### 6.3.5.3 External Validity

We can use our technique to evaluate other kinds of refactorings than the ones evaluated in this article because it does not rely on specific properties of the transformation. We just

have to change the Alloy specification in JDOLLY [77] to generate programs that exercise a specific kind of refactoring. For example, we can adapt our technique to test the Move Field refactoring by reusing our Java meta-model and well-formed rules. We need generating programs with at least two classes (*C1* and *C2*) and one field (*F1*) in one of the classes. The following Alloy fragment specifies it.

---

```
1 one sig C1, C2 extends Class DP{ }
2 one sig F1 extends Field DP{ }
3 pred generate[] {
4   F1 in C1·fields
5 }
```

---

Currently, JDOLLY generates programs considering a subset of the Java constructs. We found input programs in the test suite of Eclipse and JRRT that have some constructs, which JDOLLY does not deal with, such as anonymous class, enumeration, annotation, and synchronized. We can extend JDOLLY to consider more Java constructs and test more kinds of refactorings. In our previous work [77], we explained how we can extend the Java meta-model specified in Alloy to generate richer method bodies. We also explained the well-formed rules specified in Alloy to remove some uncompileable programs. In addition, we proposed CDOLLY, a C program generator [47]. Richer method bodies in Java can be specified similarly in Alloy to the approach used in CDOLLY to specify C functions.

### 6.3.6 Answers to the Research Questions

Next, we answer our research questions.

- **Q1** Can the DP technique detect bugs related to overly strong preconditions in the refactoring implementations?

We found a total of 30 bugs (11 new bugs) related to overly strong preconditions in 14 (70%) refactoring implementations. We did not find bugs in the Push Down Field and Rename Field refactorings of Eclipse, and Pull Up Field, Encapsulate Field, Rename Field, and Rename Type refactorings of JRRT.

- **Q2** What is the average time to find the first failure using the DP technique?

The technique can find the first bug in each refactoring implementation of JRRT on average in 0.59min. Finding the first bug in the Eclipse evaluation took on average 17min. The average time to find the first failure in Eclipse was affected by some values, such as the time to first failure in the Push Down Method and Add Parameter refactorings.

- **Q3** What is the rate of overly strong preconditions detected by the DP technique among the set of assessed preconditions?

In the refactoring implementations of Eclipse and JRRT, 60% and 62% of the evaluated preconditions in this study are overly strong, respectively.

- **Q4** Do DP and DT techniques detect the same bugs?

The techniques detect 19 bugs in common. DT technique cannot detect 11 bugs that the DP technique detected in the Add Parameter and Pull Up Field refactorings of Eclipse, and in the Move Method, Push Down Field, Rename Method, and Push Down Method refactorings of JRRT. When both refactoring engines under test have overly strong preconditions, the DT technique fails to detect bugs. The DT technique detected 5 bugs in Eclipse that the DP technique cannot detect in the Push Down Method and Rename Field refactorings of Eclipse.



# Chapter 7

## Related Work

In this chapter, we relate our work to a number of approaches for verifying and testing refactorings (Section 7.2), approaches for automated testing of refactoring engines (Section 7.2), and approaches of change impact analysis (Section 7.3).

### 7.1 Refactoring

Preconditions are a key concept of research studies on the correctness of refactorings. Opdyke [55] proposed a number of refactoring preconditions to guarantee behavior preservation. However, there was no formal proof of the correctness and completeness of these preconditions. In fact, later, Tokuda and Batory [85] showed that Opdyke’s preconditions were not sufficient to ensure behavior preservation. Roberts [64] automated the basic refactorings proposed by Opdyke.

Garrido and Johnson [19; 20] proposed CRefactory, a refactoring engine for C. They specified a set of refactoring preconditions that support programs in the presence of conditional compilation directives and implemented the refactorings. However, they did not prove them sound. We can use our technique to test their refactoring implementations with respect to overly strong preconditions. We evaluated Eclipse CDT with respect to overly weak preconditions by using DOLLY to generate C programs and SAFEREFACTOR for C. We just need to create some transformation DP, similar to the one created to Eclipse JDT, to disable preconditions of the Eclipse CDT refactoring test suite.

Kim et al. [34] conducted surveys, interviews, and quantitative analysis to evaluate refac-

toring challenges and benefits at Microsoft. Although participants of the survey mentioned that refactorings help on improving maintainability, 77% of them mentioned regression bugs as risks for applying refactorings. Also, except for the rename refactoring, most of the participants mentioned that they manually perform refactorings, despite the awareness of automated tools. This study indicates that tool support for refactoring should go beyond automated transformations. For example, they need to use a better tool support for checking behavior preservation correctness, as SAFEREFACTORIMPACT does.

Rachatasumrit and Kim [59] studied the impact of a transformation on regression tests by using the version history of Java open source projects. Among the evaluated research questions, they investigated whether the regression tests are adequate for refactorings in practice. They found that refactoring changes are not well tested: regression test cases cover only 22% of impacted entities. Moreover, they found that 38% of affected test cases are relevant for testing the refactorings. We proposed SAFEREFACTORIMPACT that uses change impact analyses to guide the test suite generation for only testing the methods impacted by a transformation. Most of the tests generated by our tool are relevant for evaluating the transformations considered in our work. Although our tool has low change coverage in larger subjects, it focuses only on generating tests to run on both versions of the program. There are a number of added or removed methods that are not exercised indirectly. So, it cannot generate tests for them.

Steimann and Thies [81] showed that by changing access modifiers (*public*, *protected*, *package*, *private*) in Java one can introduce compilation errors and behavioral changes. They proposed a constraint-based approach to specify Java accessibility, which favors checking refactoring preconditions and computing the changes of access modifiers needed to preserve the program behavior. Such specialized approach is useful for detecting bugs regarding accessibility-related properties. On the other hand, our approach is general enough for detecting bugs with respect to other OO and AO constructs.

Schäfer et al. [69] proposed refactorings for concurrent programs. They have proved the correctness based on the Java memory model. Currently, we do not deal with concurrency since SAFEREFACTORIMPACT can only evaluate sequential Java programs. However, they have demonstrated that some useful refactorings are not influenced by concurrency. In those situations, we can use SAFEREFACTORIMPACT. Later, they [68] implemented a number of

Java refactoring implementations in JRRT. They aim to improve correctness of the refactoring implementations of Eclipse. They included some of the results presented by Steimann and Thies [81]. We evaluated five JRRT implementations and found some bugs.

Tip et al. [84] presented an approach that uses type constraints to verify preconditions of those refactorings, determining which part of the code they may modify. Using type constraints, they also proposed the refactoring Infer Generic Type Arguments [17], which adapts a program to use the Generics feature of Java 5, and a refactoring to migration of legacy library classes [1]. Eclipse implemented these refactorings. Their technique allows sound refactorings with respect to type constraints. However, a refactoring may have preconditions related to other constructs. Our tool may be helpful in those situations.

Borba et al. [6] proposed a set of refactorings for a subset of Java with copy semantics (ROOL). They proved the refactoring correctness based on a formal semantics. Silva et al. [76] proposed a set of behavior preserving transformation laws for a sequential object-oriented language with reference semantics (rCOS). They proved the correctness of each of the laws with respect to rCOS semantics. Some of these laws can be used in the Java context. Yet, they have not considered all Java constructs, such as overloading and field hiding. SAFEREFACTORIMPACT may be useful when their work may not be applicable.

Li and Thompson [39] introduced a technique to test refactorings using a tool, called Quvid QuickCheck, for Erlang. They evaluated a number of implementations of the Wrangler refactoring engine. For each refactoring, they state a number of properties that it must satisfy. If a refactoring applies a transformation, but does not satisfy a property, they indicate a bug in the implementation. They found four bugs. We use SAFEREFACTOR to evaluate behavior preservation. Our technique uses a similar approach for testing refactorings for Java and C. 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 DOLLY.

Overbey and Johnson [57] proposed a technique to check for behavior preservation. They implement it in a library containing preconditions for the most common refactorings. Refactoring engines for different languages can use their library to check refactoring preconditions. The preservation-checking algorithm is based on exploiting an isomorphism between graph nodes and textual intervals. They evaluated their technique for 18 refactorings in refactoring engines for Fortran 95, PHP 5 and BC. In our approach, we use SAFEREFACTORIMPACT

to evaluate whether any transformation is behavior preserving. Proving refactorings with respect to a formal semantics constitutes a challenge [70].

In our previous work [80] we proposed a technique to identify overly strong preconditions based on differential testing [43]. If a tool correctly applies a refactoring according to SAFEREFACTOR and another tool rejects the same transformation, the latter has an overly strong precondition. In a sample of 42,774 programs generated by JDOLLY, we evaluated 27 refactorings of Eclipse, NetBeans and JastAdd Refactoring Tools (JRRT) [68], and found 17 and 7 types of overly strong preconditions in Eclipse and JRRT, respectively. This approach is useful for detecting whether the set of refactoring preconditions is minimal. Later, Soares et al. [77] introduced a technique to test refactoring tools and found more than 100 bugs in the best academic (JRRT) and commercial Java refactoring implementations (Eclipse and NetBeans). This approach is based on a program generator (JDOLLY) and SAFEREFACTOR. In this work, we extend SAFEREFACTOR to consider AO constructs, and use change impact analysis to generate tests only for the methods impacted by a transformation.

Monteiro and Fernandes [48] presented a catalog of 27 AO refactorings. They can be useful for implementing aspect-aware refactoring tools. However, they did not prove their soundness. We can apply their refactorings and use SAFEREFACTORIMPACT to improve confidence that the transformation is correct.

Wloka et al. [91] introduced a tool support for extending currently OO refactoring implementations for considering aspects. They employ change impact analysis to identify pointcuts impacted by a transformation that can change the program behavior. The tool can change pointcuts to preserve program behavior in some cases. SAFEREFACTORIMPACT does not apply a transformation to a program. It only evaluates whether a transformation preserves behavior. SAFIRA also considers aspects during the analysis. Moreover, SAFEREFACTORIMPACT evaluates any kind of transformation, while their tool evaluates only some Java refactorings, such as rename, move, extract and inline.

Binkley et al. [4; 5] presented a human guided automated approach to refactor OO to AO program. They implement six kinds of refactorings. Each refactoring defines a set of preconditions to guarantee behavior preservation. They refactored four OO real systems to modularize it in aspects (JHotDraw, PetStore, JSpider and JAccounting). Hannemann et al. [26] introduced a role-based refactoring approach to help programmers modularize crosscutting

concerns into aspects. Malta and Valente [41] presented a collection of transformations used to enable the extraction of crosscutting statements to aspects. Each refactoring defines a set of preconditions. Their work may contribute for improving tool support for applying refactorings to AO programs. However, they did not prove them sound with respect to a formal semantics. Developers can use our tool together with their approaches to improve confidence that the transformation preserves behavior. Moreover, SAFEREFACTORIMPACT can evaluate any kind of transformation.

Yokomori et al. [94] analyzed two software applications that have been refactored into aspects (JHotDraw and Berkeley DB) to determine circumstances when such activities are effective at reducing component relationships and when they are not. They found that AO refactoring is successful in improving the modularity and complexity of the base code. In our work, we propose a tool based on change impact analysis to improve confidence that a transformation preserves behavior. SAFEREFACTORIMPACT does not evaluate whether the resulting program improves the quality of the original program.

Hannemann and Kiczales [25] implemented 23 design patterns [18] in Java and AspectJ. Their study concludes that some patterns are better implemented using OO constructs and others using AO constructs. Taveira et al. [83] modularized exception handling in OO and AO code by using test suite and pair programming. The study indicates that the AO version promotes reuse of exception handling code. We used SAFEREFACTORIMPACT to analyze some transformations they evaluated, and found some behavioral changes that developers were unaware. SAFEREFACTORIMPACT does not evaluate whether the resulting program improves the quality of the original program.

Van Deursen et al. [88] used an existing well-designed open-source system (JHotDraw) and modified it to an equivalent AO version (AJHotDraw). In this work, we analyzed some transformations applied to JHotDraw collected from its SVN repository history and from studies that aimed to modularize the exception handling mechanism.

Cole and Borba [8] formally specified AO programming laws (each law defines a bidirectional semantics-preserving transformation) for AspectJ. By composing them, they derived AspectJ refactorings. Each law formally states preconditions. They proved one of them sound with respect to a formal semantics for a subset of Java and AspectJ [9]. They can be useful for implementing aspect-aware refactoring tools. However, they did not consider all

AspectJ constructs and their catalog is incomplete. In those situations, we can use our tool.

Testing refactoring correctness can be useful in other contexts as well. Dig and Johnson [11] studied the API changes of some frameworks. They have discovered that more than 80% of the changes that break API clients are refactorings. This suggests that refactoring-based migration engines should be used to update applications. API users can use SAFER-EFACTORIMPACT for checking whether API changes modify their programs' behavior. Furthermore, Reichenbach et al. [62] proposed 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.

## 7.2 Automated Testing of Refactoring Engines

Daniel et al. [10] proposed an approach for automated testing refactoring engines. The technique is based on ASTGEN, a Java program generator, and a set of programmatic oracles. To evaluate the refactoring correctness, they implemented six oracles that evaluate the output of each transformation. For instance, the oracles check for compilation errors and warning messages. There is one oracle that evaluates behavior preservation. It checks whether applying a refactoring to a program, its inverse refactoring to the target program yields the same initial program. If they are syntactically different, the refactoring engine developer has to manually check whether they have the same behavior. For example, consider the classes  $A$ ,  $B$  (subclass of  $A$ ) and  $C$  (subclass of  $B$ ) presented in Listing 7.1. The class  $A$  declares the field  $k$ , which is initialized with 10. The class  $C$  has the field  $k$  hiding  $A.k$ , which is initialized with 20, and the method  $test$  calls  $super.k$ . This method yields 10. By using Eclipse JDT 4.3 to apply the Pull Up Field refactoring to  $C.k$  moving it to class  $B$ , the transformation yields the program presented in Listing 7.2. This transformation introduces a behavioral change: the method  $test$  now calls  $B.k$  yielding 20 instead of 10 in the initial program. Applying the Push Down Field refactoring to  $B.k$  in the modified program presented in Listing 7.2, the resulted program is equal to the initial program presented in Listing 7.1. So, their oracle

does not detect this behavioral change, different from SAFEREFACTORIMPACT.

Figure 7.1: Pulling up a field introduces a behavioral change in Eclipse JDT 4.3.

Listing 7.1: Original Program	Listing 7.2: Resulting Program
<pre> <b>public class</b> A {   <b>public int</b> k = 10; } <b>public class</b> B <b>extends</b> A { } <b>public class</b> C <b>extends</b> B {   <b>public int</b> k = 20;   <b>public int</b> test() {     <b>return super.k</b>;   } } </pre>	<pre> <b>public class</b> A {   <b>public int</b> k = 10; } <b>public class</b> B <b>extends</b> A {   <b>public int</b> k = 20; } <b>public class</b> C <b>extends</b> B {   <b>public int</b> test() {     <b>return super.k</b>;   } } </pre>

They used the oracles Differential Testing, Inverse Transformations, and Custom Oracles to identify transformation issues. The Custom oracle is aware of the structural changes that their corresponding refactorings should make and thus check that the refactored program exhibits the expected changes. Our SCA oracle is based on their Custom oracle. But they did not make available the refactoring definitions used to implement this oracle. They evaluated the technique by testing 42 refactoring implementations and found three transformation issues using Differential Testing and Inverse oracles, and only one bug using the Custom oracle. We evaluated 8 refactoring implementations and found 18 transformation issues (18 using SCA oracle and 5 using DT oracle).

They identified a total of 21 bugs in Eclipse JDT and 24 in NetBeans. In Eclipse JDT, 17 bugs were related to compilation errors, 3 bugs were related to incomplete transformations (e.g. the Encapsulate field refactoring did not encapsulate all field accesses), and 1 bug was related to behavioral change. We found 17 bugs related to behavioral change in 18 refactoring implementations of JRRT and Eclipse.

Jagannath et al. [29] presented the STG technique to reduce the costs of bounded-exhaustive testing by skipping some test inputs. They randomly select a skip up to 20 after generating each program. They evaluated it using ASTGEN and found that the technique

took some seconds to find the first failure related to compilation error or engine crash in the refactoring implementations using STG. We also included the skip parameter in DOLLY to reduce the time to test the refactoring implementations and to find the first failure, which can be related to compilation error or behavioral change. Different from them we use skips to identify overly strong preconditions and transformation issues. Also, we use a fixed skip that is set by the user while they use a random skip. As our results are deterministic, we can execute the tests again using the same skip to evaluate whether we have already fixed the bugs. Moreover, we can execute using a different skip to find some missed bugs. Finally, they did not measure the rate of missed bugs using skips to generate programs different from our work.

Later, Gligoric et al. [22] proposed UDITA, a Java-like language that extends ASTGEN allowing users to describe properties in UDITA using any desired mix of filtering and generating style in opposed to ASTGEN that uses a purely generating style. UDITA evolved ASTGEN to be more expressive and easier to use, usually resulting in faster program generation as well. They found four new bugs related to compilation errors in Eclipse in a few minutes. However, the technique requires substantial manual effort for writing test generators [21] since they are specified in a Java-like language. Soares et al. [77] found that UDITA does not generate some programs that JDOLLY generates using the same scope and without skipping.

More recently Gligoric et al. [21] used real systems to reduce the effort for writing test generators using the same oracles [22]. They found 141 bugs related to compilation errors in refactoring implementations for Java and C in 285 hours. However, the technique may be costly to apply the refactorings in large systems and to minimize the failure into a small program to categorize the bugs. Moreover, evaluating transformations on large real programs is time consuming, and it would produce less accurate results. We can use SAFEREFACTORIMPACT to automatically detect behavioral changes in their technique. SAFEREFACTORIMPACT detected behavioral transformations applied on real systems that even a well-defined manual inspection conducted by experts did not detect [78; 46].

In our previous work [80; 77] we proposed a technique to test refactoring engines by detecting bugs related to compilation errors, behavioral changes, and overly strong precon-



ditions. It is based on JDOLLY, an exhaustive program generator, a set of automated oracles, such as SAFEREFACTOR [79], and differential testing to identify overly strong preconditions. As opposed to ASTGEN and UDITA that use a Java-like language, JDOLLY only needs to declaratively specify the structures of the programs. However, it may be costly to evaluate all test inputs. It took a total of 590 hours to detect 106 bugs related to compilation errors and behavioral changes in 39 refactoring implementations. Moreover, the technique does not test refactorings applied within method level. In this work, we optimize the technique to reduce the costs of testing. For example, using a skip of 25 in the program generator, it reduces in 95% the time to test the refactoring implementations while missing only 5.2% of the bugs.

Vakilian and Johnson [87] presented a technique to detect usability problems in refactoring engines. It is based on refactoring alternate paths. They adapt critical incident technique to refactoring tools and show that alternate refactoring paths are indicators of the usability problems of refactoring tools. Their technique manually found two usability problems related to overly strong preconditions. We use SAFEREFACTORIMPACT to evaluate whether the applied transformation is behavior preserving. Our technique automatically found 10 bugs related to overly strong preconditions in Eclipse JDT and JRRT.

### 7.3 Change Impact Analysis

Law and Rothermel [36] proposed an approach based on static and dynamic partitioning and recursive algorithms of calls graphs to identify methods impacted by a change. Different from SAFIRA, the analysis estimates the change impact before applying the transformation. Our change impact analyzer performs static analysis in any kind of transformation applied to Java or AspectJ programs. In addition, it does not need additional information to evaluate a transformation.

Chianti [63] is a change impact analyzer tool for Java. Based on a test suite and the changes applied to a program, it decomposes the change into atomic changes and generates a dependency graph. The tool indicates the test cases that are impacted by the change. Only these test cases need to be executed again. Zhang et al. [95] proposed a change impact analyzer tool (FaultTracer) that improves Chianti by refining the dependencies between the atomic changes, and adding more rules to calculate the change impact. Both tools re-

ceive two program versions as parameters, and decompose the change into small-grained transformations, similar to SAFIRA. However, different from SAFIRA, Chianti and Fault-Tracer depend on a test suite to assess the change impact. They execute the test cases, and identify the impacted test cases that must be executed again based on the call graphs. SAFEREFACTORIMPACT automatically generates test cases for the methods impacted by a transformation.

Kung et al. [35] presented an approach to identify impacted classes due to structural changes in library classes of OO languages. It is based on a reverse engineering approach that extracts information from the library classes and their relationships. This information is represented in dependency graphs used to automatically identify changes and their effects. Li and Offut [40] conducted a study to evaluate how changes applied to OO programs can affect program classes. They proposed an algorithm that computes the transitive closure of the program dependency graph. They analyzed changes in a program to identify impacted classes. SAFIRA also identifies the methods impacted by a change.

Wloka et al. [92] proposed a tool called JUnitMX. It uses a change impact analysis tool to yield all entities impacted by a transformation. After executing a test suite, it indicates whether the test suite exercises all entities impacted by a transformation. If all test cases pass, but they do not cover all entities impacted by a transformation, the tool yields a yellow bar. The tool yields a green bar if and only if the test cases pass and exercise all entities impacted by a transformation. Otherwise, it yields a red bar. As a future work, we intend to include this functionality in SAFEREFACTORIMPACT.

# Chapter 8

## Conclusions

In this work, we propose a technique to scale testing of refactoring engines by extending a previous technique [77; 80]. We improve it with respect to expressiveness of the program generator, reduction of costs, and detection of more kinds of bugs. We propose a new version of DOLLY with two new features: skip parameter and new Java constructs (Chapter 4). We present a strategy to reduce the time to test the refactoring implementations by skipping some consecutive test inputs [47]. Consecutive programs generated by DOLLY tend to be very similar, potentially detecting the same kind of bug. Thus, developers can set a parameter to skip some programs to reduce the time to test the refactoring implementations. By skipping those programs, we can reduce the Time to First Failure (TTFF), reducing the developer idle time. To improve the expressiveness of DOLLY we add new Java constructs, such as abstract classes and methods, and interface.

The previous technique [77; 80] uses a set of oracles to evaluate the correctness of the transformations related to overly strong preconditions, compilation errors, and behavioral changes. It uses Differential Testing (DT technique) to identify faults related to overly strong preconditions. We propose a new oracle to identify overly strong preconditions by disabling some preconditions (Chapter 6). We also refine the oracle to identify behavioral changes [46] by including a change impact analysis step (Chapter 3). Finally, we present new oracles to identify a new kind of bug related to transformation issues in refactoring implementations (Chapter 5). The oracles are based on Differential Testing (DT) and Structural Change Analysis (SCA).

We perform a more extensive evaluation to compare SAFEREFACTORIMPACT with

---

SAFEREFACTOR with respect to two new defined metrics (change coverage and relevant tests), time to evaluate the refactoring implementations, and behavioral change transformation detected. Additionally, we evaluate SAFEREFACTORIMPACT in the context of Aspects, which shows evidence that the technique is useful to evaluate transformations in AspectJ programs. SAFEREFACTORIMPACT found a number of behavioral change transformations applied by developers in real systems up to 79 KLOC. It found behavioral changes that SAFEREFACTOR cannot detect using the same time limit because SAFEREFACTORIMPACT generates much more relevant tests than SAFEREFACTOR. We can adapt our technique to use real systems as test inputs such as Gligoric et al. [21] proposed using their technique. In this case, we can use SAFEREFACTORIMPACT as an oracle to evaluate the correctness of the transformations. We only need to increase SAFEREFACTORIMPACT's timelimit.

After identifying the failures, the proposed technique uses a set of automated bug categorizers to classify all failing transformations into distinct bugs. In our previous work [80] we used a similar approach than the proposed by Jagannath et al. [29] (Oracle-based Test Clustering - OTC) to automate the classification of failures related to overly strong preconditions. The previous technique [77] uses the OTC approach to automatically classify failures related to compilation errors into distinct bugs. They specified a systematic, but manual approach to categorize failures related to behavioral changes. We automate it in this work. Also, we propose automated issue categorizers for classify the issues identified by DT and SCA oracles.

We evaluated our technique to scale testing of refactoring engines in 28 kinds of refactoring implementations of JastAdd Refactoring Tools (JRRT) [68], Eclipse JDT (Java) and Eclipse CDT (C). We found 119 bugs in a total of 49 bugs related to compilation errors, 17 bugs related to behavioral changes, 35 bugs related to overly strong preconditions using DP and DT techniques, and 18 transformation issues using SCA and DT oracles. We also compared the impact of the skip on the time consumption and bug detection in our technique. The technique reduces the time in 90% and 96% using skips of 10 and 25 in Dolly while missing only 3% and 6% of the bugs, respectively.

When using skips, the refactoring engine developer can detect a number of bugs in a few hours. The developer can run the technique again without skipping while fixing the detected bugs in order to find some missed bugs. Moreover, we can reduce even more the idle time

---

of the developer. The technique finds the first failure in the refactoring implementations in a few seconds using a skip of 10 or 25. When there are many failures transformations in a refactoring implementation, the TTFB is similar even varying the skip to generate programs. So, the developer can find a bug in a few seconds, fix the bug, run it again to find another bug, and so on. By using this strategy, the bug categorization step is no longer needed since there is only one failure in each execution. Before a new release, the developer can run the technique without skip to improve confidence that the implementation is correct.

We evaluated the new oracle to identify overly strong preconditions (DP technique) in 20 refactoring implementations of Eclipse and JRRT [68]. Among the set of 49 evaluated preconditions, we detected 30 (61%) overly strong preconditions in the refactoring implementations. So far, the developers of Eclipse confirmed 47% of them. The technique took on average from a few seconds or even minutes to find the first failure. We also compared the DP technique with our previous technique based on differential testing (DT technique) [80]. The techniques are complementary in terms of bug detection. The DP technique found 11 bugs (37% of new bugs) that the DT technique cannot detect while missed 5 bugs (21% of the bugs detected by the DT technique). However, the DP technique does not need another refactoring engine to evaluate the refactoring implementations. Additionally, the DP technique can facilitate debugging in the sense of it gives to the refactoring engine developers the set of overly strong preconditions. DT technique only yields the warning messages reported by the engine after it rejects a transformation.

We changed our techniques (DP and DT) to use some input programs of the Eclipse and JRRT refactoring test suites instead of programs generated by JDOLLY. We assessed 71 preconditions and detected 23 overly strong preconditions not detected by the developers. We did not detect 17 of them using as input the programs generated by JDOLLY. The developers did not find these overly strong preconditions because they do not seem to have a systematic strategy to detect them. Additionally, they do not have an automated oracle to check behavior preservation. We use SAFEREFACTORIMPACT as the oracle to help us in this activity.

Developers can improve their testing process to identify overly strong preconditions by using the DP and DT techniques. When it is possible, they can run the DP technique and after fixing the detected bugs, they run the DT technique to find more bugs. DT technique can show some additional changes that a refactoring engine can perform to enable applying a safe

---

transformation, such as replace *super* with *this* (or *this* with *super*) or add a cast in a field or method call. They can also run our techniques using their input programs instead of programs generated by JDOLLY, as we did in part of our evaluation. If the refactoring engine is not for Java, they can replace SAFEREFACTOR by other automated oracle to check behavior preservation. Each precondition avoids incorrect transformations. Therefore, all of them may be needed in the refactoring engines. Our technique reports to the developers a set of overly strong preconditions. After that, they can reason about their proposed preconditions to refine and slightly weak them. As a result, they improve the applicability of their refactoring implementations by avoiding the current scenario of only implementing the preconditions without evaluating them.

We evaluated our technique with respect to detect bugs related to transformation issues in eight refactoring implementations of Eclipse JDT 4.5 and JRRT. We used DOLLY with abstract classes and methods, and interface to evaluate the technique using SCA and DT oracles. The new Alloy specifications of the refactorings generated up to 1,051,608 Alloy instances. We used skip of 25 to reduce the time to test the refactoring implementations and found 18 kinds of transformation issues. In addition to reading some proposed informal refactoring definitions, we suggest executing the technique using DT oracle before implementing the SCA oracle for each refactoring type.

In summary, we scale a technique to test refactoring engines by improving limitations of previous techniques. These limitations are related to the kinds of bugs that can be detected (some techniques do not identify transformation issues [77] or overly strong preconditions [22; 21]), time consumption [77; 80], program generator (some techniques do not have an automated program generator to generate the test inputs [21; 87] or the program generator is not exhaustive [10; 22], has a costly setup [10; 22], or has a low expressiveness [77]), or some techniques need more than one refactoring engine to evaluate a refactoring implementation [80]. Our technique uses an automated and exhaustive program generator, DOLLY to generate the test inputs. We add some features in DOLLY to reduce the time to test the refactoring implementations by skipping some input programs and improve its expressiveness by adding more Java constructs. We refine the oracle to identify bugs related to behavioral changes by proposing SAFEREFACTORIMPACT and propose two new oracles to identify bugs related to transformations issues. Finally, we propose a new

technique to identify overly strong preconditions by disabling some preconditions. This new technique only needs one engine and although it has the manual step of disabling the preconditions, we automate this step by using aspects. Developers can restructure the code to enable using the aspects to also automatically disable the preconditions that we classified as particular cases.

## 8.1 Future Work

As future work, we aim at evaluating the new constructs added in DOLLY (abstract classes and methods, and interface) with respect to overly weak and overly strong preconditions. Adding new Java constructs in the Java meta-model increased the number of Alloy instances generated by the new specification using the same scope. To alleviate this problem without reducing the expressiveness of DOLLY, we aim at refining the new Alloy specification to deal with the state explosion of Alloy instances. The Java meta-model that we specified in Alloy [27] to generate programs, does not include some kinds of constructs such as the static modifier, inner classes, generics, and richer method bodies. Also, the specified scopes and constraints limit the number of entities in the generated programs. Then, we aim at specifying more language constructs in DOLLY to generate different programs as test inputs enabling to find other bugs in the refactoring implementations.

We plan to implement the oracle to detect transformation issues in more refactoring types, such as Rename (Class, Method, and Field), Push Down Field, and Add Parameter. We also intend to test the refactoring implementations of NetBeans. We evaluated the refactoring implementations with respect to transformation issues using skip of 25 to reduce the costs due to the increase number of instances generated by DOLLY with the new constructs. Using this skip, we may lose on average 3.9% of the bugs [47]. In order to find some potential missed bugs, we aim to run the experiment by starting the program generation at other instances, such as 5, 15, and 20. We also intend to measure the time to find the first failure. Finally, we aim to improve our refactoring definitions used to implement the SCA oracle.

We intend to investigate the refactoring engines test suites to identify limitations related to the input programs, assertion types, test strategies, and oracles. To identify them, we will conduct a survey to reach the refactoring engines developers (see Appendix A) and evaluate

their implementations using our technique. We altered our technique by replacing DOLLY with the input programs of the refactoring engines test suites and found 31 bugs related to compilation errors, behavioral changes, and overly strong preconditions not found by the developers. We will evaluate our technique with respect to transformation issues using the input programs of the refactoring engines test suites. Moreover, we aim at evaluating other refactoring types not evaluated by our technique using DOLLY, such as Extract Method.

As we showed in the evaluation presented in Section 3.3, SAFEREFACTORIMPACT does not detect the behavioral change in a defective refactoring because SAFIRA does not perform data flow analysis. So, SAFEREFACTORIMPACT may not generate test cases containing some getter methods that may be useful to expose the behavioral change. It has a parameter that, when enabled, allows us to consider all getter methods during the test suite generation. However, when using such option, the number of methods passed to the test suite generator may increase in some transformations. To alleviate this problem, we intend to improve the analysis performance and investigate whether it is worth including a data flow analysis in Safira. Additionally, we aim at defining other small-grained transformations to reduce the set of impacted methods identified by Safira. We are also interested in evaluating other automatic test suite generators in SAFEREFACTORIMPACT, such as EvoSuite [15] and Testful [2]. Finally, we intend to create an Eclipse plugin for SAFEREFACTORIMPACT.



# Bibliography

- [1] 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. ACM, 2005.
- [2] Luciano Baresi and Matteo Miraz. Testful: automatic unit-test generation for Java classes. In *Proceedings of the 32nd International Conference on Software Engineering, ICSE'10*, pages 281–284. ACM, 2010.
- [3] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.
- [4] 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.
- [5] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Tool-supported refactoring of existing Object-Oriented code into Aspects. *IEEE Transactions on Software Engineering*, 32(9):698–717, 2006.
- [6] Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Márcio Cornélio. Algebraic reasoning for Object-Oriented programming. *Science of Computer Programming*, 52:53–100, 2004.
- [7] Joan Box. Guinness, gosset, fisher, and small samples. *Statistical Science*, 2(1):45–52, 1987.

- 
- [8] Leonardo Cole and Paulo Borba. Deriving refactorings for AspectJ. In *Proceedings of the 4th Aspect-Oriented Software Development, AOSD'05*, pages 123–134. ACM, 2005.
- [9] 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. ACM, 2005.
- [10] 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. ACM, 2007.
- [11] Danny Dig and Ralph Johnson. The role of refactorings in API evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, ICSM'05*, pages 389–398. IEEE Computer Society, 2005.
- [12] Eclipse.org. Eclipse Project. At <http://www.eclipse.org>, 2016.
- [13] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, pages 63–86, 1996.
- [14] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Company, Inc., Boston, MA, USA, 1999.
- [15] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ESEC-FSE '11*, pages 416–419. ACM, 2011.
- [16] Robert Fuhrer, Adam Kiezun, and Markus Keller. Refactoring in the eclipse jdt: Past, present, and future. In *Workshop on Refactoring Tools at ECOOP, WRT'07*, pages 30–31, Berlin, Heidelberg, 2007.

- 
- [17] Robert Fuhrer, Frank Tip, Adam Kiezun, 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. Springer-Verlag, 2005.
- [18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2005.
- [19] Alejandra Garrido and Ralph Johnson. Refactoring C with conditional compilation. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ASE'03, pages 323–326. IEEE Computer Society, 2003.
- [20] Alejandra Garrido and Ralph Johnson. Analyzing multiple configurations of a C program. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM'05, pages 379–388. IEEE Computer Society, 2005.
- [21] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz, and Darko Marinov. Systematic testing of refactoring engines on real software projects. In *Proceedings of the 27th European conference on Object-Oriented Programming*, ECOOP '13, pages 629–653. Springer-Verlag, 2013.
- [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*, ICSE'10, pages 225–234. ACM, 2010.
- [23] John Goodenough and Susan Gerhart. Toward a theory of test data selection. *SIGPLAN Notes*, 10:493–510, 1975.
- [24] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, ESE '94, pages 970–978. Wiley, 1994.
- [25] Jan Hannemann and Gregor Kiczales. Design Pattern implementation in Java and Aspectj. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'02, pages 161–173. ACM, 2002.

- [26] Jan Hannemann, Gail Murphy, and Gregor Kiczales. Role-based refactoring of cross-cutting concerns. In *Proceedings of the 4th Aspect-Oriented Software Development, AOSD'05*, pages 135–146. ACM, 2005.
- [27] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. 2nd edition. The MIT Press, 2012.
- [28] Daniel Jackson, Ian Schechter, and Hya Shlyahter. Alcoa: the Alloy constraint analyzer. In *Proceedings of the 31st International Conference on Software Engineering, ICSE '00*, pages 730–733. IEEE Computer Society, 2000.
- [29] 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. Springer-Verlag, 2009.
- [30] Cem Kaner, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing*. John Wiley and Sons, 2001.
- [31] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.
- [32] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.
- [33] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-Oriented programming*, pages 220–242. Springer Berlin Heidelberg, 1997.
- [34] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th Foundations of Software Engineering, FSE'12*, pages 50:1–50:11. ACM, 2012.
- [35] David Kung, Jerry Gao, Pei Hsia, F Wen, Yasufumi Toyoshima, and Cris Chen. Change impact identification in Object-Oriented software maintenance. In *Proceedings of the*

- International Conference on Software Maintenance*, ICSM'94, pages 202–211. IEEE Computer Society, 1994.
- [36] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 19th International Conference on Software Maintenance*, ICSM'03, pages 308–318. IEEE Computer Society, 2003.
- [37] Gary Leavens, Albert Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31:1–38, 2006.
- [38] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. Contract driven development = test driven development - writing test cases. In *Proceedings of the Joint Meeting of the European Software Engineering Conference*, ESEC-FSE '07, pages 425–434. ACM, 2007.
- [39] 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.
- [40] Li Li and Jeff Offutt. Algorithmic analysis of the impact of changes to Object-Oriented software. In *Proceedings of the International Conference on Software Maintenance*, ICSM'96, pages 171–184. IEEE Computer Society, 1996.
- [41] Marcelo Malta and Marco Valente. Object-Oriented transformations for extracting Aspects. *Information and Software Technology*, 51(1):138–149, January 2009.
- [42] Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin Rinard. An evaluation of exhaustive testing for data structures. Technical report, MIT Computer Science and Artificial Intelligence Laboratory Report MIT -LCS-TR-921, 2003.
- [43] William Mckeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

- 
- [44] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30:126–139, 2004.
- [45] Melina Mongiovi. Uma Abordagem para Avaliar Refatoramentos Baseada no Impacto da Mudança. Master’s thesis, Federal University of Campina Grande, 2013.
- [46] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. Making refactoring safer through impact analysis. *Science of Computer Programming*, 93:39–64, 2014.
- [47] Melina Mongiovi, Gustavo Mendes, Rohit Gheyi, Gustavo Soares, and Márcio Ribeiro. Scaling testing of refactoring engines. In *IEEE International Conference on Software Maintenance and Evolution*, ICSME’14, pages 371–380. IEEE Computer Society, 2014.
- [48] 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. ACM, 2005.
- [49] Gail Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23:76–83, 2006.
- [50] 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, 2012.
- [51] 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. IEEE Computer Society, 2009.
- [52] Glenford Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 2004.
- [53] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP’13, pages 552–576, Berlin, Heidelberg, 2013. Springer-Verlag.

- [54] NetBeans.org. NetBeans IDE. At <http://www.netbeans.org/>, 2016.
- [55] William Opdyke. *Refactoring Object-Oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
- [56] William Opdyke and Ralph Johnson. Refactoring: An aid in designing application frameworks and evolving Object-Oriented systems. In *Proceedings of the Object-Oriented Programming emphasizing Practical Applications*, SOOPPA '90, pages 145–160, 1990.
- [57] Jeffrey Overbey and Ralph 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. ACM, 2011.
- [58] Carlos Pacheco, Shuvendu K. Lahiri, Michael Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE'07, pages 75–84. IEEE Computer Society, 2007.
- [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. IEEE Computer Society, 2012.
- [60] Henrique Rebêlo, Ricardo Lima, Márcio Cornélio, Gary T. Leavens, Alexandre Cabral Mota, and César Oliveira. Optimizing JML features compilation in Ajmle using Aspect-Oriented refactorings. In *Proceedings of the 13rd Brazilian Symposium on Programming Languages*, SBLP'09, pages 117–130. Brazilian Computer Society, 2009.
- [61] Henrique Rebêlo, Sérgio Soares, Ricardo Lima, Leopoldo Ferreira, and Márcio Cornélio. Implementing Java modeling language contracts with AspectJ. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing*, SAC'08, pages 228–233. ACM, 2008.
- [62] 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.

- [63] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. 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. ACM, 2004.
- [64] Don Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.
- [65] 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. IEEE Computer Society, 2011.
- [66] Ana Rocha, José Maldonado, and Kival Weber. *Qualidade de software: teoria e prática*. Prentice Hall, 2001.
- [67] Max Schäfer. *Specification, implementation and verification of refactorings*. PhD thesis, University of Oxford, 2010.
- [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. ACM, 2010.
- [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. Springer-Verlag, 2010.
- [70] Max Schäfer, Torbjörn Ekman, and Oege Moor. Challenge proposal: verification of refactorings. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, PLPV'09, pages 67–72. ACM, 2008.
- [71] Max Schäfer, Torbjörn Ekman, and Oege Moor. Sound and extensible renaming for Java. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'08, pages 277–294. ACM, 2008.



- [72] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Refactoring Java programs for flexible locking. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE'11, pages 71–80. ACM, 2011.
- [73] Max Schäfer, Andreas Thies, Friedrich Steimann, and Frank Tip. A comprehensive approach to naming and accessibility in refactoring Java programs. *IEEE Transactions on Software Engineering*, 38(6):1233–1257, 2012.
- [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. Springer-Verlag, 2009.
- [75] Samuel Shapiro and Martin Wilk. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, 52(3/4):591–611, 1965.
- [76] 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. IEEE Computer Society, 2008.
- [77] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39:147–162, 2013.
- [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*, 86(4):1006–1022, 2013.
- [79] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making program refactoring safer. *IEEE Software*, 27:52–57, July 2010.
- [80] 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, 2011.
- [81] 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. Springer-Verlag, 2009.
- [82] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE'76, pages 492–497. IEEE Computer Society Press, 1976.
- [83] Júlio Taveira, Cristiane Queiroz, Rômulo Lima, Juliana Saraiva, Sérgio Soares, Hítalo Oliveira, Nathalia Temudo, Amanda Araújo, Jefferson Amorim, Fernando Castor, and Emanuel Barreiros. Assessing intra-application exception handling reuse with Aspects. In *Proceedings of the 23rd Brazilian Symposium on Software Engineering*, SBES'09, pages 22–31. IEEE Computer Society, 2009.
- [84] Frank Tip, Adam Kiezun, 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. ACM, 2003.
- [85] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8:89–120, 2001.
- [86] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Proceedings of the 13th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '07, pages 632–647. Wiley, 2007.
- [87] Mohsen Vakilian and Ralph E. Johnson. Alternate refactoring paths reveal usability problems. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1106–1116. ACM, 2014.
- [88] Arie van Deursen, Marius Marin, and Leon Moonen. AJHotDraw: A showcase for refactoring to aspects. In *Proceedings of the Workshop on Linking Aspect Technology and Evolution*, LATE'05, 2005.
- [89] Frank Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

- 
- [90] Visser Willem, Pasareanu Corina, and Pelanek Radek. Test input generation for java containers using state matching. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '06. ACM, 2006.
- [91] 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. ACM, 2008.
- [92] Jan Wloka, Einar W. Host, and Barbara G. Ryder. Tool support for change-centric test development. *IEEE Software*, pages 66–71, 2010.
- [93] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Programming Language Design and Implementation*, PLDI'11, pages 283–294. ACM, 2011.
- [94] Reishi Yokomori, Harvey Siy, Norihiro Yoshida, Masami Noro, and Katsuro Inoue. Measuring the effects of Aspect-Oriented refactoring on component relationships: two case studies. In *Proceedings of the 10th Aspect-Oriented Software Development*, AOSD'11, pages 215–226. ACM, 2011.
- [95] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. FaultTracer: a change impact and regression fault analysis tool for evolving Java programs. In *Proceedings of the 20th ACM SIGSOFT Foundations of Software Engineering*, FSE'12, pages 40:1–40:4. ACM, 2012.
- [96] Hong Zhu, Patrick Hall, and John May. Software unit test coverage and adequacy. *ACM Computing Survey*, 29:366–427, 1997.

## **Appendix A**

# **Survey – Implementing and Testing Refactorings**

# Implementing and Testing Refactorings

The goal of the survey is to understand the process of implementing and testing refactorings. It consists of three parts:

- i. Participant background (4 questions)
- ii. Implementation of refactorings (14 questions)
- iii. Evaluation of the implemented refactorings (6 questions)

Answering the survey should take around 10 minutes of your time. All the data collected from the survey is anonymous. The results of the survey may be reported in academic publications. If you have any questions or concerns, please contact Melina Mongiovi <[melina@copin.ufcg.edu.br](mailto:melina@copin.ufcg.edu.br)>.

Thanks,

Melina Mongiovi, Federal University of Campina Grande, Brazil  
Rohit Gheyi, Federal University of Campina Grande, Brazil  
Sarah Nadi, University of Alberta, Canada  
Márcio Ribeiro, Federal University of Alagoas, Brazil

\* Required

## The following is the terminology that we use in the rest of the survey

---

1. Refactoring transformation: a transformation which has a goal of refactoring a program
2. Refactoring test case: a unit test included with the source code of the refactoring tool
3. Input program: program used as input by a refactoring test case
4. Resulting program: the output program produced after applying a refactoring transformation
5. Refactoring definition: structural changes that a specific refactoring is supposed to do
6. Transformation issues: incorrect transformations regarding the refactoring transformation definition
7. Refactoring conditions: conditions implemented in the refactoring tool, which goal is to prevent incorrect transformations
8. Overly weak conditions: refactoring conditions that allow to apply some incorrect transformations, which can introduce compilation errors or behavioral changes
9. Overly strong conditions: refactoring conditions that prevent applying some correct transformations

## Participant Background

### 1. Which refactoring tool do/did you work on? \*

*Mark only one oval.*

- Eclipse
- IntelliJ
- JRRT
- NetBeans

**2. For how long have you been involved as a committer to the above refactoring tools?**

*Mark only one oval.*

- Less than a year
- 1-3 years
- 4-6 years
- 7-10 years
- More than 10 years

**3. How many years of Java programming experience do you have?**

*Mark only one oval.*

- Less than a year
- 1-3 years
- 4-6 years
- 7-10 years
- More than 10 years

**4. How many years of software testing experience do you have?**

*Mark only one oval.*

- Less than a year
- 1-3 years
- 4-6 years
- 7-10 years
- More than 10 years

## Refactorings Specification

**5. Do you have any document (besides the source code) that specifies what the refactoring transformations are supposed to do?**

*Mark only one oval.*

- Yes
- No *Skip to question 9.*

## Refactorings Specification

**6. Could you provide it for us (through a link or email <[melina@copin.ufcg.edu.br](mailto:melina@copin.ufcg.edu.br)>)?**

.....

## Refactorings Specification

**7. Does the implementation follow the specification of the refactoring transformations?***Mark only one oval.*

- The implementation closely follows the specification *Skip to question 9.*
- The implementation slightly follows the specification *Skip to question 8.*
- The implementation does not follow the specification *Skip to question 8.*
- I do not know *Skip to question 9.*

**Refactorings Specification****8. Why does the implementation not follow the specification?**

.....

.....

.....

**Implementation of refactorings****9. How do you determine that a refactoring transformation is correct? You can choose more than one answer.***Check all that apply.*

- The resulting program compiles
- The resulting program has the same observable behavior of the original program
- The transformation improves the quality of the original program
- The transformation follows the expected refactoring transformation definition
- Other: .....

**10. Do you think in some cases the refactoring tool can apply transformation that introduces compilation errors or behavioral changes? Please, if you say "yes", justify your answer.**

.....

.....

.....

.....

**11. How do you specify the conditions necessary to prevent incorrect refactoring transformations? You can choose more than one answer.***Check all that apply.*

- Based on my knowledge in refactoring transformations
- Based on the reported bugs
- Based on refactoring books
- Based on refactoring papers
- Bases on the Java Specification Language
- Other: .....

12. How do you evaluate whether two versions of a program have the same observable behavior? You can choose more than one answer.

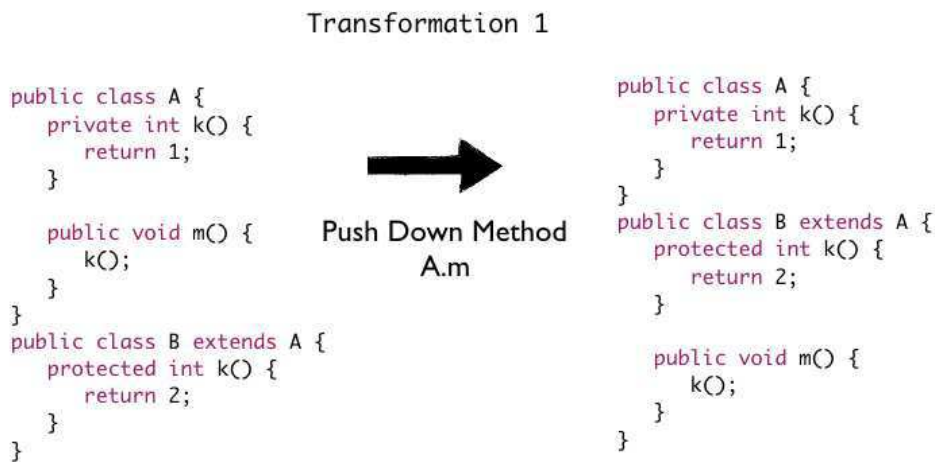
Check all that apply.

- By compiling the resulting program
- By performing a manual evaluation
- By using a tool to evaluate behavior preservation
- Other: .....

13. Does Transformation 1 preserve the program behavior?

Mark only one oval.

- Yes Skip to question 15.
- No



### Implementation of refactorings

14. Why does Transformation 1 change the program behavior?

.....

.....

.....

.....

.....

### Implementation of refactorings

15. Does Transformation 2 preserve the program behavior?

Mark only one oval.

- Yes
- No

16. Mark only one oval.

- Option 1




## Transformation 2

```

class Super {
  int k() {
    return 10;
  }
}

class A extends Super {
  int k() {
    return 20;
  }
  int m() {
    return super.k();
  }
  int test() {
    return m();
  }
}

```

  
 Pull Up Method  
 A.m

```

class Super {
  int k() {
    return 10;
  }
  int m() {
    return k();
  }
}

class A extends Super {
  int k() {
    return 20;
  }
  int test() {
    return m();
  }
}

```

## Implementation of refactorings

17. Why does Transformation 2 change the program behavior?

.....

.....

.....

## Implementation of refactorings

18. Does Transformation 3 preserve the program behavior?

Mark only one oval.

- Yes     *Skip to question 20.*
- No


## Transformation 3

```

package p;
class A<T> {
  void a( T t){
  }
}

class B extends A<String> {
  public void m(){
    super.a(null);
    super.a(new String());
  }
}

```

  
 Pull Up Method  
 B.m

```

package p;
class A<T> {
  void a( T t){}

  public void m() {
    this.a(null);
    this.a(new String());
  }
}

class B extends A<String> {}

```

## Implementation of refactorings

**19. Why does Transformation 3 change the program behavior?**

.....

.....

.....

.....

**Evaluation of the implemented refactorings**

**20. Rate the following kinds of bugs on a scale of 1 to 5, with 1 being the least important to test and 5 being the most important.**

*Mark only one oval per row.*

	1	2	3	4	5
Compilation errors	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Behavioral changes	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Transformation issues	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Overly strong conditions	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Usability problems	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
System crash	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

**21. How do you select the input programs of the test cases for your refactoring engine? You can choose more than one answer.**

*Check all that apply.*

- Based on my experience
- Based on expert opinion concerning refactoring and testing
- Based on the reported bugs
- Using lines of code, branch or instruction coverage metrics
- Using grammar coverage metric
- Other: .....

**22. How do you determine that a bug is duplicated? You can choose more than one answer.**

*Check all that apply.*

- By manually analyzing the source code to understand the fault related to each bug
- By using an automated tool
- By analyzing the compiler messages: different messages refer to different bugs
- By analyzing the program structure: different structures refer to different bugs
- By analyzing the messages reported by the engine when it rejects transformations: different messages refer to different bugs
- Other: .....

**23. Do you use any automated tool to generate the input programs of the test cases?**

.....

**24. Do you use any automated tool to check for behavior preservation?**

.....

**25. Do you use any automated tool to divide the identified failures into distinct bugs?**

## **Additional comments**

**26. Please, let us know if you have any additional comments about the process of implementing and testing refactorings.**

.....

.....

.....

.....

**27. If you would like to receive the results of our survey, please leave your email address.**

.....

