A Multi-Level Framework for the Detection, Prioritization and Testing of Software Design Defects

by

Dilan Sahin

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Information Systems Engineering)
in the University of Michigan Dearborn
2016

Doctoral Committee:

Assistant Professor Marouane Kessentini, Chair
Professor William Grosky
Assistant Professor Bochen Jia
Associate Professor Bruce Maxim

DEDICATION

This thesis work is dedicated to my dear family who has been always with me at every important step of my life. I want to thank my dear mother, Dudu Sahin, who always encouraged me to be a strong woman and be able to stay on my own foot. As a working mother, she did everything in an extraordinary way to raise me and my brother and always showed us how valuable is to be a hard-worker and fighter in this life. I owe everything I have to my strong mother. Her wish was to see me on the stage while getting my PhD diploma. How lucky I am to give her what she wished for. My father, Mahmut Sahin, is the person who gave me strength to follow my dreams. He is my number one supporter in this life. All the sacrifices he did for me to gain this experience are countless. I cannot forget the day when we get on the plane together to start my childhood dream. He is my hero. My brother, Cem Deniz Sahin, is the dearest of our family. He is my friend whom I can share everything in my life. I will always remember his cooking for me whenever I needed to study. I want to thank my dear aunt, Dondu Serin, for being my second mom whenever I needed her during my school years. She also has a lot of influence on me to make my dream come true. I also want to thank all my family members back in Turkey.

ACKNOWLEDGEMENTS

Deciding to pursue a PhD degree was not an easy choice especially when it involved me leaving everyone behind and come to United States on my own. It has been a difficult journey for me with ups and downs however it was totally worth it. This was not only an education, but a life experience which made me who I am. I became a strong person who knows what she wants from life and know how to get it if she puts enough hard work and sprinkle a bit enthusiasm in it. This achievement would not be possible if Professor Kessentini was not in the picture. I want to thank him for being the most important part of this journey. As an advisor, he gave me an opportunity to learn how to do research and introduced me to this existing field which I will continue building my career on top of it. I cannot thank him enough for his endless support, time and energy to make this dream come true. I would also want to thank my dissertation committee, Prof. Bruce Maxim, Prof. William Grosky and Prof. Bochon Jia who has been there each step I took in this journey.

I want to give my special thanks to another strong woman in my life, Emily Wang, for all her support whenever I came across an obstacle in this journey. She gave me strength, courage and hope when I most needed it. Thank you Emily.

I want to thank the members of SBSE lab whom I shared most of this journey with. I specifically want to thank Mohamed Wiem Mkaouer for his countless support during my PhD, Nivin Tama for a great collaboration for our paper, Usman Mansoor for making us laugh all the time during our lab meetings. I also want to thank my dearest friend John Baluch for his great support in the last leap of this journey. I want to thank Kate Markotan for her support at every obstacle I had during my studies.

I want to thank my friends Don Barbarci, Gaurav Sheth, Marinus Koelman, Josselin Dea, Paty Ortiz, Nick Crooker and Scott Riopelle for all their great support.

PREFACE

The research that led to this thesis was performed at Search-Based Software Engineering Laboratory at the Department of Computer and Information Science, University of Michigan-Dearborn, with Prof. Marouane Kessentini as the main advisor. This work was funded by Search-Based Software Engineering Laboratory in collaboration with Ford Motor Company.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Large-scale software systems exhibit high complexity and become difficult to maintain. In fact, it has been reported that software cost dedicated to maintenance and evolution activities is more than 80% of the total software costs. In particular, object-oriented software systems need to follow some traditional design principles such as data abstraction, encapsulation, and modularity. However, some of these non-functional requirements can be violated by developers for many reasons such as inexperience with object-oriented design principles, deadline stress. This high cost of maintenance activities could potentially be greatly reduced by providing automatic or semi-automatic solutions to increase system's comprehensibility, adaptability and extensibility to avoid bad-practices.

The detection of refactoring opportunities focuses on the detection of bad smells, also called antipatterns, which have been recognized as the design situations that may cause software failures indirectly. The correction of one bad smell may influence other bad smells. Thus, the order of fixing bad smells is important to reduce the effort and maximize the refactoring benefits. However, very few studies addressed the problem of finding the optimal sequence in which the refactoring opportunities, such as bad smells, should be ordered. Few other studies tried to prioritize refactoring opportunities based on the types of bad smells to determine their severity. However, the correction of severe bad smells may require a high effort which should be

optimized and the relationships between the different bad smells are not considered during the prioritization process.

The main goal of this research is to help software engineers to refactor large-scale systems with a minimum effort and few interactions including the detection, management and testing of refactoring opportunities. We report the results of an empirical study with an implementation of our bi-level approach. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than existing techniques based on a benchmark of 9 open source systems and 1 industrial project. We have also evaluated the relevance and usefulness of the proposed bi-level framework for software engineers to improve the quality of their systems and support the detection of transformation errors by generating efficient test cases.

KEYWORDS

Search-based Software Engineering, Model-Driven Engineering, Refactoring, Bi-level Optimization

Chapter 1: Introduction

1.1 Research Context

The key to successful software is to integrate software maintenance step into software life cycle smoothly. The recent studies have showed the importance of the software maintenance in software life cycle, total cost of the software projects, and easiness in the flexibility and extensibility of software design. Software maintenance has been the focal point of many research topics and software companies how to find better, more automated ways to increase the impact of software maintenance on software quality and also decrease the cost of software maintenance since it consumes up to 90% of the total cost of a software system.

Restructuring the internal structure of the software systems while keeping the external behavior unchanged, has been recognized as an important and powerful approach to improve the quality of the software in terms of many aspects, e.g., software maintainability, reusability, and extensibility [1, 2]. This approach has been known as Software Refactoring [2, 3] which is an effective method to increase the internal quality of software systems and widely acknowledged among software's best practices. The most important step in software refactoring is to find how to identify refactoring opportunities in the systems which may be in need of software restructuring. Code smells, also called antipatterns[4], defects, smells [5], anomalies[6] or bad design practices[7] which have been recognized as the design situations that may cause software

failures indirectly, can be summarized as the identification methods to find out which software systems or which part of software are in need refactoring[5, 8]. A code-smell is defined as bad design choices that can have a negative impact on the code quality such as maintainability, changeability and comprehensibility which could introduce bugs[12]. Code-smells classify shortcomings in software that can decrease software maintainability. They are also defined as structural characteristics of software that may indicate a code or design problem that makes software hard to evolve and maintain, and trigger refactoring of code[6]. Code-smells are not limited to design flaws since most of them occur in code and are not related to the original design. In fact, most of code-smells can emerge during the evolution of a system and represent patterns or aspects of software design that may cause problems in the further development and maintenance of the system. As stated by [4], code-smells are unlikely to cause failures directly, but may do it indirectly. In general, they make a system difficult to change, which may in turn introduce bugs. It is easier to interpret and evaluate the quality of systems by identifying code-smells than the use of traditional software quality metrics. In fact, most of the definitions of code-smells are based on situations that are daily faced by developers. Most of the code-smells identify locations in the code that violate object-oriented design heuristics, such as the situations described by Riel[13] and Coad et al.[14]. The 22 Code Smells identified and defined informally by Fowler et al. [4] aim to indicate software refactoring opportunities and 'give you indications that there is trouble that can be solved by a refactoring'. Zhang et al.[15] identified in their survey the code-smells that attracted more attention in current literature.

Van Emden and Moonen[16] developed one of the first automated code-smell detection tools for Java programs. Mantyla studied the manner of how developers detect and analyse code-

smells[17]. Previous empirical studies have analysed the impact of code-smells on different software maintainability factors including defects[18] and effort[19]. In fact, software metrics (quality indicators) are sometimes difficult to interpret and suggest some actions (refactoring) as noted by Anda et al.[20] and Marinescu et al.[20, 21]. Code-smells are associated with a generic list of possible refactorings to improve the quality of software systems. In addition, Yamashita et al. [22] show that the different types of code-smells can cover most of maintainability factors[12]. Thus, the detection of code-smells can be considered as a good alternative of the traditional use of quality metrics to evaluate the quality of software products. Brown et al. [23] define another category of code-smells that are documented in the literature, and named anti-patterns.

In our experiments, we focus on the seven following code-smell types:

- *Blob*: It is found in designs where one large class monopolizes the behavior of a system (or part of it), and the other classes primarily encapsulate data.
- *Feature Envy (FE)*: It occurs when a method is more interested in the features of other classes than its own. In general, it is a method that invokes several times accessor methods of another class.
- *Data Class (DC)*: It is a class with all data and no behavior. It is a class that passively store data
- *Spaghetti Code (SC)*: It is a code with a complex and tangled control structure.
- *Functional Decomposition (FD)*: It occurs when a class is designed with the intent of performing a single function. This is found in a code produced by non-experienced object-oriented developers.

- *Lazy Class (LC)*: A class that is not doing enough to pay for itself.
- *Long Parameter List* (*LPL*): Methods with numerous parameters are a challenge to maintain, especially if most of them share the same data-type.

We choose these code-smell types in our experiments because they are the most frequent, hard to detect and fix based on recent empirical study[24], cover different maintainability factors, and also due to the availability of code-smell examples. However, the proposed approach in this thesis is generic and can be applied to any type of code-smells. The code-smell detection process consists in finding code fragments that violate structural or semantic properties such as the ones related to coupling and complexity. In this setting, internal attributes used to define these properties, are captured through software metrics, and properties are expressed in terms of valid values for these metrics. This follows a long tradition of using software metrics to evaluate the quality of the design including the detection of code-smells. The most widely-used metrics are the ones defined by Chidamber and Kemerer[25] and other studies[12, 26]. In this thesis, we use variations of these metrics and adaptations of procedural ones as well[27, 28]. The list of metrics is described in Table I.

On the other hand, model transformation testing is considered as one of the main challenges in MDE [32]. Figure 1 illustrates the model transformation pattern extended for the context of model transformation testing. In order to test a given transformation implementation, a set of test input models is needed for running the transformation to obtain test output models. As soon as the output models of the transformation runs are available, the validity of the output models can be determined by using an oracle function. In the context of model transformations, model comparison may be used by defining the expected output models for given input models or

contracts may be applied that are evaluated over pairs of input and output models [33-35].

In this thesis, we focus on the generation of test input models. Thus, we reuse existing approaches for specifying oracle functions and transformations implementations in the context of this work.

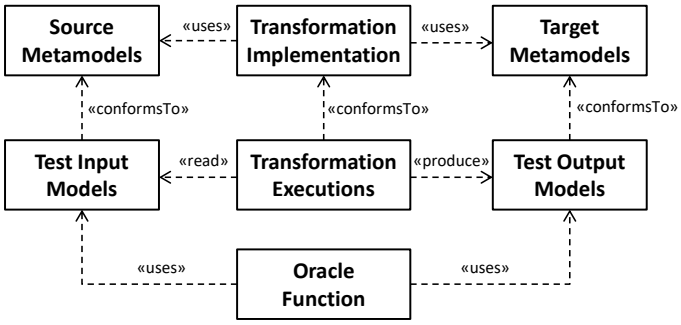


Figure 1 Building blocks of model transformation testing

1.1.1.1 Model Transformation Example

Before we proceed with discussing the relationships between test case generation and the analysis of the generated test cases, we introduce a small transformation example to explicate the definition of metamodels, models, and model transformations. The example we use for this purpose is the Families to Person example1 presented in the ATL documentation example collection. The source and target metamodels and models are illustrated in UML class diagram and UML object diagram notation, respectively, in Figure 2. In this example, the family members are transformed into persons based on their family/member relationships. Furthermore, the fullNames of persons are computed based on the concatenation of the firstName of persons and the lastName of the families. An example source/target model pair is used to exemplify these

[1] http://www.eclipse.org/atl/documentation/basicExamples_Patterns/

transformation requirements.

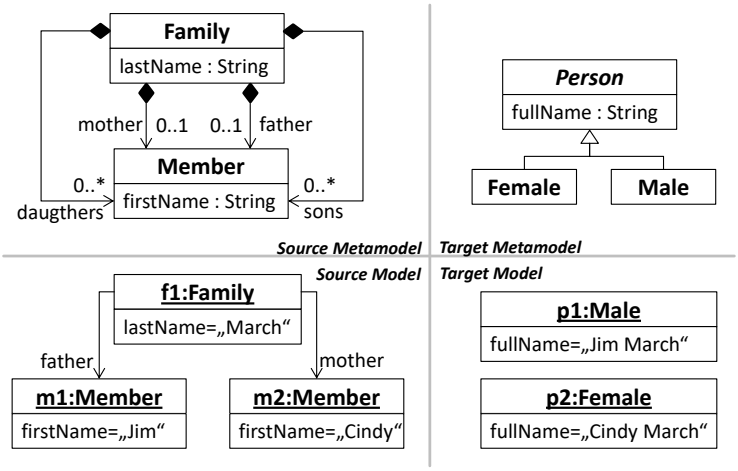


Figure 2 Family to Person example: metamodels and example models.

In Listing 1, the corresponding ATL code for the given transformation example can be found. The transformation module consists of two helpers and two rules. Helpers are used to define reusable queries. Helpers are called from the rules which are the major elements of an ATL transformation. A rule consists of a part which is querying elements from the source models and of a to part which is generating elements in the target models and setting their features by defining bindings (cf. <- operator in Listing 1). By running this transformation for the given source model of Figure 2, the corresponding target model is produced. An example for an oracle function, to detect a possible error, may be the following requirement for the transformation: there have to be as many Person instances in the target model as we have Member instances in the source model. This may be true for the given sample source model, but for other source models, the opposite may be the case. Thus, proper means for generating test source models is needed to judge if a transformation is actually realizing the stated requirements in terms of oracle functions or not.

Listing 1: Family to Person example: ATL transformation.

```
module FamiliesToPersons;
create OUT: Pers from IN: Fam;

helper context Fam!Member def: isFemale : Boolean =...

helper context Fam!Member def: familyName : String =...

rule Member2Male {
  from s:Fam!Member (not s.isFemale)
  to t: Pers!Male(
    fullName <- s.firstName+' '+s.familyName
  )
}

rule Member2Female {
  from s:Fam!Member (s.isFemale)
  to t:Pers!Female(
    fullName <- s.firstName+' '+s.familyName
  )
}
```

1.2 Problem Statement

In tackling increasing software degradation, the first thing to look is where exactly the problematic areas in the software are, in other words, to detect the design defects in the system. There have been proposed many solutions to detect those anomalies in the software system, however most of these solutions involve detecting the code smells manually. Those methods are mostly based on declarative rule specification in which the identification of the code-smells are made by the manually defined rules. Hence, code smells are identified by combinations of mainly quantitative (metrics), structural, and/or lexical information. However, manual code-smell identification process has some drawbacks. However, in an exhaustive scenario, the number of possible code-smells used to manually characterize with rules can be large. For each code-smell, rules that are expressed in terms of metric combinations need substantial calibration efforts to find the right threshold value for each metric. Another important issue is that translating symptoms into rules is not obvious because there is no consensual symptom-based

definition of code-smells[9]. When consensus exists, the same symptom could be associated to many code-smells types, which may compromise the precise identification of code-smell types. These difficulties explain a large portion of the high false-positive rates reported in existing research. Recently, a Search-Based Software Engineering (SBSE) approach[10], based on genetic programming[11], is used to generate code-smell's detection rules from a set of examples of code-smells identified manually by developers[7]. However, such approaches require a high number of code-smell examples (data) to provide efficient detection rules solutions. In fact, code-smells are not usually documented by developers (unlike bugs report). Thus, it is time-consuming and difficult to collect code-smells and inspect manually large systems. In addition, it is challenging to ensure the diversity of the code-smell examples to cover most of the possible bad-practices.

As the last step, we want to emphasize the importance of testing. In the current state of work, there is a lack of generating the optimum test cases which suit the specific system the most. With that inspiration, as our last step, we try to tackle this problem.

In this thesis, we are addressing how to overcome the current problems by bringing the bi level optimization approach into the picture, and getting highly good results after applying detection, maintenance, refactoring and testing steps and showing the importance how these steps are highly dependent on each other. Next sections we give a detailed description about the current problems in the existing work that our contribution is trying to solve.

Table 1 List of quality metrics

Metrics	*Description*
Weighted Methods per Class (WMC)	WMC represents the sum of the complexities of its methods.
Response for a Class (RFC)	RFC is the number of different methods that can be executed when an object of that class receives a message.
Lack of Cohesion of Methods (LCOM)	Chidamber and Kemerer define Lack of Cohesion in Methods as the number of pairs of methods in a class that does not have at least one field in common minus the number of pairs of methods in the class that does share at least one field. When this value is negative, the metric value is set to 0.
Number of Attributes (NA)	
Attribute Hiding Factor (AH)	AH measures the invisibilities of attributes in classes. The invisibility of an attribute is the percentage of the total classes from which the attribute is not visible.
Method Hiding Factor (MH)	MH measures the invisibilities of methods in classes. The invisibility of a method is the percentage of the total classes from which the method is not visible.
Number of Lines of Code (NLC)	NLC counts the lines but excludes empty lines and comments.
Coupling Between Object classes (CBO)	CBO measures the number of classes coupled to a given class. This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.
Number of Association (NAS)	
Number of Classes (NC)	
Depth of Inheritance Tree (DIT)	DIT is defined as the maximum length from the class node to the root/parent of the class hierarchy tree and is measured by the number of ancestor classes. In cases involving multiple inheritances, the DIT is the maximum length from the node to the root of the tree.
Polymorphism Factor (PF)	PF measures the degree of method overriding in the class inheritance tree. It equals the number of actual method overrides divided by the maximum number of possible method overrides.
Attribute Inheritance Factor (AIF)	AIF is the fraction of class attributes that are inherited.
Number of Children (NOC)	NOC measures the number of immediate descendants of the class.

1.2.1 Code Smells Detection

In this section, we introduce some issues and challenges related to the detection of code-smells. Overall, there is no general consensus on how to decide if a particular design violates a quality heuristic. In fact, there is a difference between detecting symptoms and asserting that the detected situation is an actual code-smell. For example, an object-oriented program with a hundred classes from which one class implements all the behavior and all the other classes are only classes with attributes and accessors. No doubt, we are in the presence of a Blob. Unfortunately, in real-life systems, we can find many large classes, each one using some data classes and some regular classes. Deciding which classes are Blob candidates heavily depends on the interpretation of each analyst. In some contexts, an apparent violation of a design principle may be consensually accepted as normal practice. For example, a “Log” class responsible for maintaining a log of events in a program, used by a large number of classes, is a common and acceptable practice. However, from a strict code-smell definition, it can be considered as a class with an abnormally large coupling. Another issue is related to the definition of thresholds when dealing with quantitative information. For example, the Blob detection involves information such as class size. Although we can measure the size of a class, an appropriate threshold value is not trivial to define. A class considered large in a given program/community of users could be considered average in another. All the above issues make the process of manually defining code-smell detection rules challenging. The generation of detection rules requires a huge code-smell example set to cover most of the possible bad-practice behaviors. Code-smells are not usually documented by developers (unlike bugs report). Thus, it is time-consuming and difficult to collect code-smells and inspect manually large systems [13]. In addition, it is challenging to

ensure the diversity of the code-smell examples to cover most of the possible bad-practices then using these examples to generate good quality of detection rules.

We address the open issues related to applying code smells detection approaches.

Problem 1.2.1.1 The vast majority of existing work in code-smells detection relies on declarative rule specification [7, 29, 30]. In these settings, rules are manually defined to identify the key symptoms that characterize a code-smell using combinations of mainly quantitative (metrics), structural, and/or lexical information. However, in an exhaustive scenario, the number of possible code-smells used to manually characterize with rules can be large.

Problem 1.2.1.2 For each code-smell, rules that are expressed in terms of metric combinations need substantial calibration efforts to find the right threshold value for each metric.

Problem 1.2.1.3 Another important issue is that translating symptoms into rules is not obvious because there is no consensual symptom-based definition of code-smells [7, 31]. When consensus exists, the same symptom could be associated to many code-smells types, which may compromise the precise identification of code-smell types.

Problem 1.2.1.4 Recently, Search-Based Software Engineering (SBSE) approach, based on genetic programming is used to generate code-smells detection rules from a set of examples of code-smells identified manually by developers [7]. However, such approaches require a high number of code-smell examples (data) to provide efficient detection rules solutions. In

fact, code-smells are not usually documented by developers (unlike bugs report). Thus, it is time-consuming and difficult to collect code-smells and inspect manually large systems.

Problem 1.2.1.5 It is challenging to ensure the diversity of the code-smell examples to cover most of possible bad-practices.

1.2.2 Model Transformation Testing

1.2.2.1 Open Issue: Marrying Generation & Analysis of Test Input Models

For the generation of test models, several different techniques (e.g.,[36-39] to mention just a few) are available that are based on different notions of coverage criteria. While some approaches focus mainly on black-box criteria such as metamodel coverage (e.g., which meta-classes and which meta-features are actually instantiated by the test models), other approaches focus on white-box criteria such as transformation coverage (e.g., rule, statement, or path coverage). These approaches support model generator components (cf. upper part of Figure 3) that are mostly based on constraint satisfaction techniques such as CSP or SAT. In particular, they transform the metamodels and/or model transformations to constraints which are solved in order to provide the requested test data.

Of course, the coverage criteria is only an approximation for the fitness of a set of test input models. In order to determine the effectiveness of a test set, mutation-based analysis of test sets for model transformations have been proposed[40, 41] and applied in several studies to compare the appropriateness of different coverage criteria, e.g., see [42].

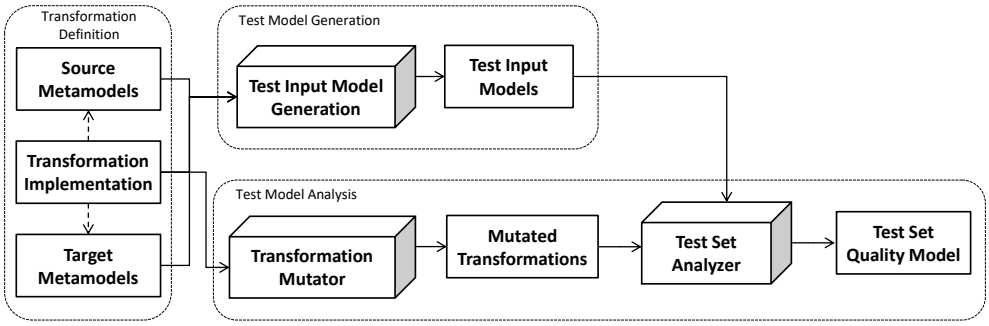


Figure 3 Uni-directional relationship between test model generation and analysis

In the context of model transformations, mutation testing is achieved by defining a set of mutation operators based on fault models, i.e., typical errors that occur, for model transformations which are applied to the model transformation implementation. The result of applying the mutation operators to a model transformation is called a mutant (cf. lower part of Figure 3). The mutant is killed if the test input models are able to detect the mutation, i.e., one test should fail. The goal is to kill as many mutations as possible for a given set of input models. For instance, a mutant of the transformation in Listing 1 may be produced by changing the type of the output element (cf. to part) of the second rule from Female to Male or by deleting the bindings for setting the fullName values in the output models. Mutation-based analysis is mostly used as a post-generation activity for quality assurance as can be seen in Figure 3. The model generation phase feeds the analysis phase, but not the other way round. However, we propose to consider the relationship between generation and analysis as a bi-directional one. Instead of "just" evaluating the quality, the evaluation results may be again propagated to the model generation component in order to improve automatically the set of test input models.

1.3 Proposed Solutions

To address the above mentioned problems, we propose the following solutions which are organized into four principal contributions. First of all, we want to talk about Bi-Level optimization since in all our contributions it is the main part.

1.3.1 Contribution 1: Bi-Level Code Smells Detection

In this thesis, we introduce a novel formulation of the code-smells detection as a bi-level problem. We report the results of an empirical study with an implementation of our bi-level approach. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than existing techniques based on a benchmark of 9 open source systems and 1 industrial project. We also evaluate the relevance and usefulness of the detected code-smells for software engineers to improve the quality of their systems.

Contribution 1.1 We have proposed a bi-level evolutionary optimization approach. The upper-level optimization produces a set of detection rules, which are combinations of quality metrics, with the goal to maximize the coverage of not only a code-smell example base but also a lower-level population of artificial code-smells.

Contribution 1.2 The lower-level optimization tries to generate artificial code-smells that cannot be detected by the upper-level detection rules, thereby emphasizing the generation of broad-based and fitter rules.

Contribution 1.3 The statistical analysis of the obtained results over nine studied software systems have shown the competitiveness and the outperformance of our proposal in terms of

precision and recall over a single-level genetic programming, co-evolutionary, and non-search-based methods.

1.3.2 Contribution 2: Model Transformation Testing

A novel formulation of model transformation testing as a bi-level optimization problem is introduced. We report the results of an empirical study with an implementation of our bi-level approach. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than an existing technique based on metamodels coverage.

Contribution 2.1 In our adaptation, the upper level generates a set of test cases (models) which maximizes the coverage of source and target metamodels; and mutants (errors introduced in the rules/program) are generated by the lower level.

Contribution 2.2 The lower level maximizes the number of errors that cannot be detected by test cases produced by the upper level. The errors are detected by comparing the target models produced by both programs (rules) at the lower (mutants) and upper levels. If they are the same, we can assume that the test cases were not sufficient to cover all the transformation possibilities.

Contribution 2.3 The overall problem appears as a BLOP task, where for each generated test cases, the upper level observes how the lower-level acts by generating errors/mutants that cannot be detected by the upper level (leader) test cases, and then chooses the best test cases which suits it the most, taking the actions of the errors generation process (lower level or follower) into account.

Contribution 2.4 The main advantage of our bi-level formulation is that the generation of test cases is not limited to the metamodels coverage but it allows evaluating the ability of generated test cases to detect errors and their coverage of transformation possibilities.

Contribution 2.5 In addition, we considered in our approach the dependency between mutation analysis and test cases generation and we do not consider them as two separate steps. We implemented our proposed bi-level approach and evaluated it on two different ATL transformation programs.

Chapter 2: Related Work

2.1 Bi-Level Optimization Technique

Most studied real-world and academic optimization problems involve a single level of optimization. However, in practice, several problems are naturally described in two levels. These latter are called BLOPs[43]. In such problems, we find a nested optimization problem within the constraints of the outer optimization one. The outer optimization task is usually referred as the *upper level problem* or the *leader problem*. The nested inner optimization task is referred as the *lower level problem* or the *follower problem*, thereby referring the bi-level problem as a leader-follower problem or as a Stackelberg game[44]. The follower problem appears as a constraint to the upper level, such that only an optimal solution to the follower optimization problem is a possible feasible candidate to the leader one. A BLOP contains two classes of variables: (1) the upper level variables $x_u \in X_U \subset \Re^n$ and (2) the lower level variables $x_l \in X_L \subset \Re^m$. For the follower problem, the optimization task is performed with respect to the variables x_l , and the variables x_u act as fixed *parameters*. Thus, each x_u corresponds to a *different* follower problem, whose optimal solution is a function of x_u and needs to be determined. All variables (x_u, x_l) are considered in the leader problem, but x_l are not changed (cf. Figure 1). In what follows, we give the formal definition of BLOP:

Definition 1: Assuming $L : \Re^n \times \Re^m \to \Re$ to be the leader problem and $f : \Re^n \times \Re^m \to \Re$ to be the follower one, analytically, a BLOP could be stated as follows:

$$\underset{x_u \in X_U, x_l \in X_L}{Min} L(x_u, x_l) \text{ subject to } \begin{cases} x_l \in ArgMin\{f(x_u, x_l), g_j(x_u, x_l) \le 0, j = 1,...,J\} \\ G_k(x_u, x_l) \le 0, k = 1,...,K \end{cases}$$

BLOPs are intrinsically more difficult to solve than single-level problems, it is not surprising that most of existing studies to date has tackled the simplest cases of BLOPs, i.e., problems having nice properties such as linear, quadratic or convex objective and/or constraint functions. In particular, the most studied instance of BLOPs has been for a long time is the linear case in which all objective functions and constraints are linear with respect to the decision variables.

Although the first works on bi-level optimization date back to the seventies, it was not until the early eighties that the usefulness of these mathematical programs in modeling hierarchical decision processes and engineering problems prompted researchers to pay close attention to BLOPs. A first bibliographical survey on the subject was written by Kolstad [43] in mid-eighties. BLOPs being intrinsically more difficult than single-level problems, it is not surprising that most algorithmic research to date has tackled the simplest cases of BLOPs, i.e., problems having nice properties such as linear, quadratic or convex objective and/or constraint functions[45, 46]. In particular, the most studied instance of BLOPs has been for a long time the linear case in which all objective functions and constraints are linear with respect to the decision variables[47, 48].

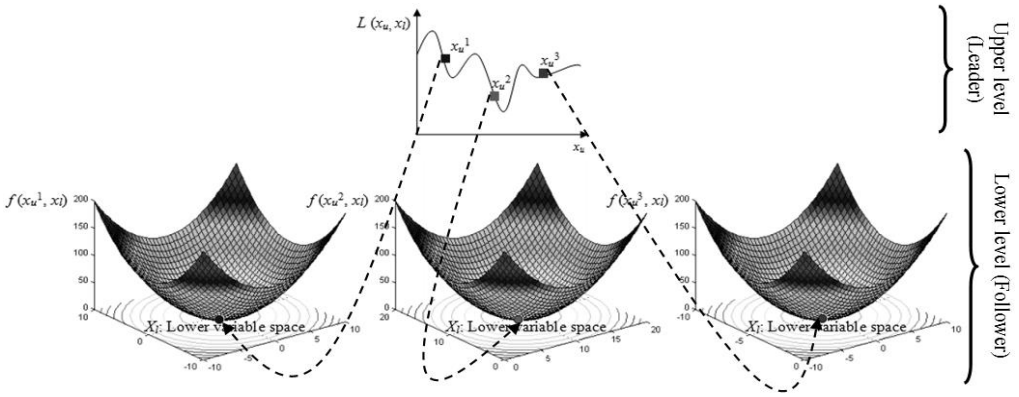


Figure 4 Illustration of the two levels of an exemplified bi-level single-objective optimization problem.

Existing methods to solve BLOPs could be classified into two main families: (1) classical methods and (2) evolutionary methods. The first family includes extreme point-based approaches[49], branch-and-bound[46], complementary pivoting[50], descent methods[51], penalty function methods [51], trust region methods[52], etc. The main shortcoming of these methods is that they heavily depend on the mathematical characteristics of the BLOP at hand. The second family includes meta-heuristic algorithms that are mainly Evolutionary Algorithms (EAs). Recently, several EAs have demonstrated their effectiveness in tackling such type of problems thanks to their insensibility to the mathematical features of the problem in addition to their ability to tackle large-size problem instances by delivering satisfactory solutions in a reasonable time. Some representative works are by [44, 53, 54].

To the best of our knowledge and based on recent surveys [55, 56], there is no work in the software engineering literature that considers a software engineering problem as a bi-level one.

2.2 Code-Smells Detection

The vast majority of existing work in code-smells detection relies on declarative rule specification[17]. In these settings, rules are manually defined to identify the key symptoms that characterize a code-smell using combinations of mainly quantitative (metrics), structural, and/or lexical information. However, in an exhaustive scenario, the number of possible code-smells used to manually characterize with rules can be large. For each code-smell, rules that are expressed in terms of metric combinations need substantial calibration efforts to find the right threshold value for each metric. Another important issue is that translating symptoms into rules is not obvious because there is no consensual symptom-based definition of code-smells[9]. When consensus exists, the same symptom could be associated to many code-smells types, which may compromise the precise identification of code-smell types. These difficulties explain a large portion of the high false-positive rates reported in existing research. Recently, a Search-Based Software Engineering (SBSE) approach[64], based on genetic programming[97], is used to generate code-smell's detection rules from a set of examples of code-smells identified manually by developers[7]. However, such approaches require a high number of code-smell examples (data) to provide efficient detection rules solutions. In fact, code-smells are not usually documented by developers (unlike bugs report). Thus, it is time-consuming and difficult to collect code-smells and inspect manually large systems. In addition, it is challenging to ensure the diversity of the code-smell examples to cover most of the possible bad-practices. There are several studies that have recently focused on detecting code-smells in software using different techniques. These techniques range from fully automatic detection to guided manual inspection.

2.2.1 Interactive-based approaches

In [4], Fowler and Beck have described a list of design smells that may exist in the program. They suggested that the software maintainers should manually inspect the program to detect existing design smells. In addition, they specify particular refactorings for each code-smell type. Travassos et al. [16] have also proposed a manual approach for detecting code-smells in object-oriented designs. The idea is to create a set of "reading techniques" that help a reviewer to "read" a design artifact for finding relevant information. These reading techniques give specific and practical guidance for identifying code-smells in object-oriented design. In this way, each reading technique helps the maintainer focusing on some aspects of the design, in such a way that the inspection team applying the entire family should achieve a high degree of coverage of the design code-smells. In addition, in [57], another proposed approach is based on violations of design rules and guidelines. This approach consists of analyzing legacy code, specifying frequent design problems as queries and locating the occurrences of these problems in a model derived from the source code. The high rate of false-positives generated by the above-mentioned approaches encouraged other teams to explore semi-automated solutions. These solutions took the form of visualization-based environments. The primary goal is to take advantage of the human capability to integrate complex contextual information in the detection process. Kothari et al. [58] present a pattern-based framework for developing tool to detect software anomalies by representing potential code-smells with different colors. Dhambri et al. [59] have proposed a visualization-based approach to detect design anomalies by automatically detecting some symptoms and letting others to a human analyst. The visualization metaphor was chosen specifically to reduce the complexity of dealing with a large amount of data. Although

visualization-based approaches are efficient to examine potential code-smells on their program and in their context, they do not scale to large systems easily. In addition, they require great human expertise, and thus they are still time-consuming and error-prone strategies. Moreover, the information visualized is mainly metric-based, meaning that complex relationships can be difficult to detect. Indeed, since visualization approaches and tools such as VERSO [59] are based on manual and human inspection, they are still, not only, slow and time-consuming, but also subjective.

The main disadvantage of existing manual and interactive-based approaches is that they are ultimately a human-centric process which requires a great human effort and strong analysis and interpretation effort from software maintainers to find design fragments that correspond to code-smells. In addition, these techniques are time-consuming, error-prone and depend on programs in their contexts.

2.2.2 Symptom-based detection

Moha et al. [60] started by describing code-smell symptoms using a domain-specific-language (DSL) for their approach called DECOR. They proposed a consistent vocabulary and DSL to specify anti-patterns based on the review of existing work on design code-smells found in the literature. Symptom descriptions are later mapped to detection algorithms. However, converting symptoms into rules needs a significant analysis and interpretation effort to find the suitable threshold values. In addition, this approach uses heuristics to approximate some notions, which results in an important rate of false positives. Indeed, this approach has been evaluated on only four well-known design code-smells: the Blob, functional decomposition, spaghetti code, and

Swiss-army knife because the literature provides obvious symptom descriptions on these code-smells. Recently, another probabilistic approach has been proposed by Khomh et al. [30] extending the DECOR approach [60], a symptom-based approach, to support uncertainty and to sort the code-smell candidates accordingly using a Bayesian Belief Network (BBN). The detection outputs are probabilities that a class is an occurrence of a code-smell type, i.e., the degree of uncertainty for a class to be a code-smell. They also showed that BBNs can be calibrated using historical data from both similar and different context. Similarly, Munro et al.[8] have proposed a template-based approach using a precise definition of bad smells from the informal descriptions given by the originators Fowler and Beck [4]. The template consists of three main parts: a code-smell's name, a text-based description of its characteristics, and heuristics for its detection.

In another category of work based on the use quality metrics, Marinescu et al. [17] have proposed a mechanism called "detection strategy" for formulating metrics-based rules that capture deviations from good design principles and heuristics. Detection strategies allow to a maintainer to directly locate classes or methods affected by a particular design code-smell. As such, Marinescu has defined detection strategies for capturing around ten important flaws of object-oriented design found in the literature. After his suitable symptom-based characterization of design code-smells, Salehie et al.[61] proposed a metric-based heuristic framework to detect and locate object-oriented design flaws similar to those illustrated by Marinescu [17]. It is accomplished by evaluating design quality of an object-oriented system through quantifying deviations from good design heuristics and principles by mapping these design flaws to class level metrics such as complexity, coupling and cohesion by defining rules. Erni et al. [53]

introduce the concept of multi-metrics, as an n-tuple of metrics expressing a quality criterion (e.g., modularity). Unfortunately, multi-metrics neither encapsulate metrics in a more abstract construct, nor do they allow a flexible combination of metrics.

Van Emden and Moonen[16] developed one of the first automated code-smells detection tools for Java programs. They developed a prototype code-smells browser that detects and visualizes code-smells in Java programs. Mantyla studied the manner of how developers detect and analyse code-smells[9]. Previous empirical studies have analyzed the impact of code-smells on different software maintainability factors [18]. In fact, software metrics (quality indicators) are sometimes difficult to interpret and suggest some actions (refactoring) as noted by Anda et al.[18] and Marinescu et al.[21]. In addition, Yamashita et al.[22] show that the different types of code-smells can cover different maintainability factors[12]. In other studies, Yamashita et al.[62] analyzed the benefits of detecting code-smells on several industrial projects and evaluated their impact on different maintainability aspects. Recently, Palomba et al. [24] used the history of changes to detect code-smells instead of the use of quality metrics. For example, classes that change frequently are considered as a blob. A comparison with DECOR confirms the outperformance of their approach and the benefits of the use of the history of changes for the detection of code-smells.

2.2.3 Search-based Approaches

SBSE [10] uses search-based approaches to solve optimization problems in software engineering. After the formulation of software engineering task as a search problem, many search algorithms can be applied to solve that problem. In [63], we have proposed another

approach, based on search-based techniques, for the automatic detection of potential code-smells in code. We used the notion that the more code deviates from good practices, the more likely it is bad. In another work [7], we generated detection rules defined as combinations of metrics/thresholds that better conform to known instances of bad-smells (examples). Then, the correction solutions, a combination of refactoring operations, should minimize the number of bad-smells detected using the detection rules. Thus, our previous work treats the detection and correction as two different steps.

Based on recent SBSE surveys[63, 64], the use of bi-level optimization is still very limited in software engineering. Indeed, this work represents the first attempt to use bi-level optimization to address software engineering problem.

2.3 Model Transformation Testing

Several techniques have been proposed to generate test cases for model transformations [32, 33, 37, 38, 40, 104]. The majority of them generate models conforming to source metamodels in order to maximize the coverage of metamodels. With respect to the contribution of this thesis, we organize related work into two categories. First, we survey approaches used for test input model generation, and second, we elaborate on approaches used for mutation-based analysis of test input models. We base our related work categorization on recent surveys on model transformation testing approaches.

2.3.1 Test Input Model Generation

The generation of test input models for model transformations may be distinguished into two

general approaches: black-box based approaches and white-box based approaches. Regarding white-box-based methods in the area of model transformations, Küster et al. [36] assume the existence of a high-level design of model transformations to produce test cases. Consequently, to apply this approach to existing model transformations such as ATL transformations, the manual extraction of these conceptual transformation rules is required, being in contrast to our vision of testing these transformations directly and automatically. In Gonzáles & Cabot [38] a very interesting white-box based testing approach for ATL transformations is provided by extracting OCL constraints from ATL code to automate the generation of test input models. The generation of test cases from Triple Graph Grammars is discussed in [30].

Besides white-box testing, many approaches have been proposed for black-box testing, i.e., test source models may be generated either on basis of the source metamodels as done in [9,10,20], the specified requirements, for instance, as visual contracts [14], as done in [13,21], or both [19]. For the actual test source model generation, most of these approaches rely–similar to software engineering–on constraint satisfaction, e.g., by means of SAT solvers. Finally, the Tracts approach [22,23,44] provides a semi-automatic approach, since the transformation designer must specify a generation script on basis of the declarative ASSL language [11]. Other approaches are using fully manually engineered test input and output models, especially ones that are using model comparison as oracle function [17].

Finally, in [15] we presented an approach to test model transformations based on existing input/output model pairs that are equipped with trace links. By analyzing trace links produced by new transformation runs, one may detect errors in transformation implementations when having untypical trace links for newly transformed models. Regression testing for model transformations

when metamodels evolve based on search-based techniques has been discussed in [27]. Another work [28] discusses the application of a bacteriologic algorithm [29] (an extension of genetic algorithm) to the generation of test cases in order to find an optimal solution with respect to metamodel coverage.

To sum up, none of the aforementioned approach has considered the application of mutation-based analysis for actively guiding the generation of test input models.

2.3.2 Mutation-based Analysis of Transformations

The application of mutation analysis for a set of test input models to predict the quality of the test set is elaborated in several publications [18,24,31]. The first two papers discuss different mutation operators for model transformations. Regarding the application of mutation-based analysis of model transformations test data, different coverage criteria (mostly black-box ones) have been evaluated by means of mutation testing (cf., e.g., [13,20]). In [31] coverage criteria are defined based on mutation testing for programmed graph transformations. The only work we are aware of combining mutation analysis with the adaption of test cases is presented in [25], however there are two main differences compared to our work. First, in our approach we introduce mutants more guided by having the competition between the two levels of our bi-level approach instead of generating mutations fully randomly, and second, we do not only use as stopping criterion that 100% of mutations are detected, because this may be also due to the fact that the introduced mutants are easy to detect independent of how good the test cases are.

Chapter 3: Code-Smells Detection as a Bi-Level Problem

3.1 Introduction

In this work, we start from the observation that the generation of efficient code-smells detection rules heavily depends on the coverage and the diversity of the used code-smell examples. In fact, both mechanisms for the generation of detection rules and the generation of code-smell examples are dependent. Thus, the intuition behind this work is to generate examples of code-smell that cannot be detected by some possible detection rules solutions then adapting these rules-based solutions to be able to detect the generated code-smell examples. These two steps are repeated until reaching a termination criterion (e.g. number of iterations). To this end, we propose, for the first time, to consider the code-smell's detection problem as a bi-level one [52]. Bi-Level Optimization Problems (BLOPs) are a class of challenging optimization problems, which contain two levels of optimization tasks[43]. In these problems, the optimal solutions to the lower level problem become possible feasible candidates to the upper level problem. In our adaptation, the upper level generates a set of detection rules, combination of quality metrics, which maximizes the coverage of the base of code-smell examples; and artificial code-smells are generated by the lower level. The lower level maximizes the number of generated “artificial” code-smells that cannot be detected by the rules produced by the upper level. The overall problem appears as a BLOP task, where for each generated detection rule, the upper level observes how the lower-level acts by generating artificial code-smells that cannot be detected by the upper level (leader) rule, and then chooses the best detection rule which suits it the most, taking the actions of the code-

smells generation process (lower level or follower) into account. The main advantage of our bi-level formulation is that the generation of detection rules is not limited to some code-smell examples identified manually by developers that are difficult to collect but it allows the prediction of new code-smell behaviours that are different from those in the base of examples.

We implemented our proposed bi-level approach and evaluated it on several open source systems: JFreeChart, GanttProject, ApacheAnt, Nutch, Log4J, Lucene, Xerces-J, and Rhino. In addition, we evaluated our proposal on one industrial system provided by our industrial partner, i.e., the Ford Motor Company. We found that, on average, the majority of seven types of code-smells were detected with more than 86% of precision and 90% of recall. The statistical analysis of our experiments over 31 runs shows that BLOP performed significantly better than two existing search-based approaches[7, 98] and a practical code-smell detection technique [60]. The software developers considered in our experiments confirm the relevance of the detected code-smells for several maintenance activities.

The primary contributions of this chapter can be summarized as follows:

(1) The chapter introduces a novel formulation of the code-smell's detection as a bi-level problem.

(2) The chapter reports the results of an empirical study with an implementation of our bi-level approach. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than existing techniques based on a benchmark of 9 open source systems and 1 industrial project. The chapter also evaluates the

relevance and usefulness of the detected code-smells for software engineers to improve the quality of their systems.

3.2 Approach

This section shows how the above-mentioned issues can be addressed using bi-level optimization and describes the principles that underlie the proposed method for detecting code-smells. We first present an overview of our bi-level code-smells detection approach, and then we describe the details of our bi-level formulation including the adaptation of both lower and upper levels. The concept of bi-level is based on the idea that the main optimization task is usually termed as the upper level problem, and the *nested* optimization task is referred to as the lower level problem. The detection rules generation process has a main objective which is the generation of detection rules that can cover as much as possible the code-smells in the base of examples. The code-smell's generation process has one objective that is maximizing the number of generated artificial code-smells that cannot be detected by the detection rules and that are dissimilar from the base of well-designed code examples. There is a hierarchy in the problem, which arises from the manner in which the two entities operate. The detection rules generation process has higher control of the situation and decides which detection rules for the code-smells generation process to operate in. Therefore, in this framework, we observe that the detection rules generation process acts as a leader (the important output of the problem), and the code-smells generation process acts as a follower.

The overall problem appears as a bi-level optimization task, where for each generated detection rules, the upper level observes how the lower-level acts by generating artificial code-smells that cannot be detected by the upper level (leader) rules and then chooses the best detection rules which suit it the most, taking the actions of the code-smells generation process (lower level or follower) into account. It should be noted that in spite of different objectives appearing in the problem, it is not possible to handle such a problem as a simple multi-objective optimization task. The reason for this is that the leader cannot evaluate any of its own strategies without knowing the strategy of the follower, which it obtains only by solving a nested optimization problem.

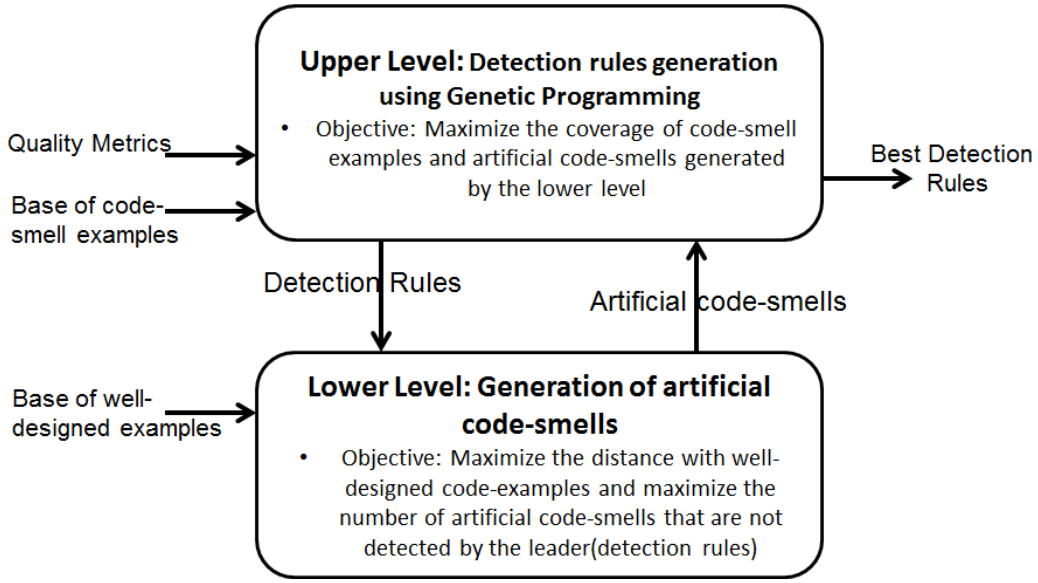


Figure 5 Approach Overview

As described in Figure 2, the leader (upper level) uses knowledge from code-smells examples (input) to generate detection rules based on quality metrics (input). It takes as inputs a base (i.e. a set) of code-smells' examples, and takes, as controlling parameters, a set

of quality metrics and generates as output a set of rules. The rule generation process chooses randomly, from the metrics provided list, a combination of quality metrics (and their threshold values) to detect a specific code-smell. Consequently, the solution is a set of rules that best detect the code-smells of the base of examples. For example, the following rule states that a class *c* having more than *NAD=10* attributes and more than *NMD=20* methods is considered as a blob smell:

R1: IF *NAD(c)≥10* AND *NMD(c)≥20,* THEN *Blob(c).*

In this exemplified sample rule, the number of attributes (NAD) and the number of methods (NMD) of a class correspond to two quality metrics that are used to detect a blob. The detected code-smells can be method(s) or class(es) depending on the type of the code-smells to detect. Given the above detection rule, it is not obvious what set of diverse code-smells exist in the software. In the bi-level formulation of the code-smell detection problem, the lower level problem allows us to find just that. An upper-level detection rules solution is evaluated based on the coverage of the base of code-smell examples (input) and also the coverage of generated "artificial" code-smells by the lower-level problem. These two measures are used to be maximized by the population of detection rules solutions. The follower (lower level) uses well-designed code examples to generate "artificial" code-smells based on the notion of deviation from a reference (well-designed) set of code fragments. The generation process of artificial code-smell examples is performed using a heuristic search that maximizes on one hand, the distance between generated code-smell examples and

reference code examples and, on the other hand, maximizes the number of generated examples that are not detected by the leader (detection rules).

There is no parallelism in our bi-level formulation. The upper level is executed for number iterations then the lower level for another number of iterations. After that the best solution found in the lower level will be used by the upper level to evaluate the associated solution (detection rules), and then this process in repeated several times until reaching a termination criterion (e.g. number of iterations). Thus, there is no parallelism since both levels are dependent. In [98], we proposed to use Co-Evolutionary (Co-Evol) algorithms for code-smells detection where the first population generates detection rules and the second one generates artificial code-smell. Both populations are executed in parallel *without hierarchy*. The problem with the Co-Evol approach is that one population may converge before the other. Contrariwise, in our bi-level approach there is a *hierarchy* that allows avoiding the problem of premature convergence of one population over the other. Indeed, the evaluation of every detection rule solution (upper level) requires the running a search algorithm to find the best undetectable artificial code-smells by the upper level solution. This concept avoids driving the search towards *uninteresting* directions. In addition, co-evolution treats the two populations independently; however in BLOP, the evaluation of solutions in the upper level depends on the lower level (both populations cannot be executed in parallel). Furthermore, the two populations in co-evolution are considered with the same importance; however the upper level is more important than the lower level in any bi-level formulation. We will compare later in the experimentation section, with more details, the difference between our

bi-level formulation for code-smells detection and our previous work based on co-evolution[98].

Next, we describe our adaptation of bi-level optimization to the code-smells detection problem in more details.

3.2.1 Problem Formulation

The code-smell's detection problem involves searching for the best metric combinations among the set of candidate ones, which constitute a huge search space. A solution of our code-smells detection problem is a set of rules (metric combinations with their threshold values) where the goal of applying these rules is to detect code-smells in a system.

Our proposed bi-level formulation of the code-smell detection problem is described in Figure 3. Consequently, we have two levels as described in the previous section. At the upper level, the objective function is formulated to maximize the coverage of code-smell examples (input) and also maximize the coverage of the generated artificial code-smells at the lower level (best solution found in the lower level). Thus, the objective function at the upper level is defined as follows:

$$Maximize \ \ f_{upperLevel} = \frac{\frac{\Pr ecision(S, baseOfExamples) + \mathrm{Re} call(S, BaseOfExamples)}{2} + \frac{\#\det ectedArtificialCodeSmells}{\#artificialCodeSmells}}{2}$$

It is clear that the evaluation of solutions (detection rules) at the upper level depends on the best solutions generated by the lower level (artificial code-smells). Thus, the fitness

function of solutions at the upper level is calculated after the execution of the optimization algorithm in the lower level at each iteration.

At the lower level, for each solution (detection rule) of the upper level an optimization algorithm is executed to generate the best set of artificial code-smells that cannot be detected by the detection rules at the upper level. An objective function is formulated at the lower level to maximize the number of un-detected artificial code-smells that are generated and also maximize the distance with well-designed code-examples. Formally,

$$Maximize \quad f_{lower} = t + Min\left(\sum_{j=1}^{w}\sum_{k=1}^{l}\left|M_k(cArtificialCS) - M_k(c\,\mathrm{Re}\,ferenceCode)\right|\right)$$

where w is the number of code elements (e.g. classes) in the reference code, l is the number of structural metrics used to compare between artificial code-smells and the well-designed code examples, M is a structural metric (such as number of methods, number of attributes, etc.) and t is the number of artificial code-smells uncovered by the detection rule solution defined at the upper level.

Upper level algorithm: GPSmellDetection

01. **Inputs:** Quality metrics *M*, Defect example base *B*, Well-designed code example base *D*, Number of best upper solutions that are considered for lower level optimization *nbs*, Upper population size N_1, Lower population size N_2, Upper number of generations G_1, Lower number of generations G_2
02. **Output:** Best detection rule BDR
03. **Begin**
04. $P_0 \leftarrow$ Initialization (N_1, *M*);
05. **For each** DR_0 **in** P_0 **do** /**DR means Detection Rule*/*

06. $BCS_0 \leftarrow$ GASmellGeneration (DR_0, D, N_2, G_2); */*Call lower level*

07. $DR_0 \leftarrow$ Evaluation (DR_0, B, BCS_0);

08. **End For**

09. $t \leftarrow 1$;

10. **While** ($t < G1$) **do**

11. $Q_t \leftarrow$ Variation (P_{t-1});

12. **For each** DR_t **in** Q_t **do** */*Evaluate each rule based on upper fitness function*/*

13. $DR_t \leftarrow$ UpperEvaluation (DR_t, B);

14. **End For**

15. **For each of the best** *nbs* **rules** DR_t **in** Q_t **do** */*Only nbs rules are used to*/*

16. $BCS_t \leftarrow$ GASmellGeneration (DR_t, D, N_2, G_2);

17. $DR_t \leftarrow$ EvaluationUpdate (DR_t, BCS_t); */*Update based on lower level*/*

18. **End For**

19. $U_t \leftarrow P_t \cup Q_t$;

20. $P_{t+1} \leftarrow$ EvironmentalSelection (N_1, U_t);

21. $t \leftarrow t+1$;

22. **End While**

23. $BDR \leftarrow$ FittestSelection (P_t);

24. **End**

Figure 6 (a) Pseudo-code of the bi-level adaptation for code-smell's detection

Lower level algorithm: GASmellGeneration

01. **Inputs:** Upper level detection rule *UDR*, Well-designed code example base *D*, Population size *N*, number of generations *G*

02. **Output:** Best artificial code-smells *BCS*

03. **Begin**

04. $P_0 \leftarrow$ Initialization (N, D);

05. $P_0 \leftarrow$ Evaluation (P_0, D, *UDR*); /**Evaluation depends of UDR**/

06. $t \leftarrow 1$;

07. **While** ($t < G$) **do**
08. $Q_t \leftarrow$ Variation (P_{t-1});
09. $Q_t \leftarrow$ Evaluation (Q_t, *D, UDR*); /**Evaluation depends of UDR*/
10. $U_t \leftarrow P_t \cup Q_t$;
11. $P_{t+1} \leftarrow$ EvironmentalSelection (N, U_t);
12. $t \leftarrow t+1$;
13. **End While**
14. *BCS* $\leftarrow$ FittestSelection (P_t);
15. **End**

Figure 7 (b) Pseudo-code of the bi-level adaptation for code-smell's detection

3.2.2 Solution Approach

The solution approach proposed in this chapter lies within the SBSE field. As noted by Harman et al.[10], a generic algorithm like bi-level optimization cannot be used 'out of the box' – it is necessary to define problem-specific genetic operators to obtain the best performance. To adapt bi-level optimization to our code-smell's detection problem, the required steps are to create for both levels (algorithms): (1) solution representation, (2) solution variation and (3) solution evaluation. We examine each of these in the coming paragraphs.

3.2.2.1 Solution Representation

One key issue when applying a search-based technique is to find a suitable mapping between the problem to solve and the techniques to use, i.e., detecting code-smells.

For the *upper-level* optimization problem, a genetic programming (GP) algorithm is used[11]. In GP, a solution is composed of terminals and functions. Therefore, when

applying GP to solve a specific problem, they should be carefully selected and designed to satisfy the requirements of the current problem. After evaluating many parameters related to the code-smells detection problem, the terminal set and the function set are decided as follows. The terminals correspond to different quality metrics with their threshold values (constant values). The functions that can be used between these metrics are Union (OR) and Intersection (AND). More formally, each candidate solution *S* in this problem is a sequence of detection rules where each rule is represented by a binary tree such that:

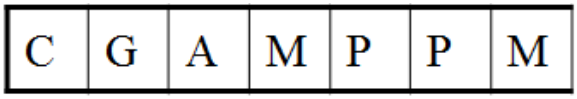
(1) each leaf-node (Terminal) L belongs to the set of metrics (such as number of methods, number of attributes, etc.) and their corresponding thresholds generated randomly.

(2) each internal-node (Functions) *N* belongs to the Connective (logic operators) set C = AND, OR.

The set of candidate's solutions (rules) corresponds to a logic program that is represented as a forest of AND-OR trees.

For the *lower-level* optimization problem, a genetic algorithm (GA) is used to generate artificial code-smells. The generated artificial code fragments are composed of code elements. Thus, they are represented as a vector where each dimension is a code element. We represent these elements as sets of predicates. Each predicate type corresponds to a construct type of an object-oriented system: *Class (C), attribute (A), method (M), parameter (P), generalization (G), and method invocation relationship between classes (R)*. For example, the sequence of predicates *CGAMPPM* corresponds to a class with a generalization link,

containing two attributes and two methods (Figure 4). The first method has two parameters. Predicates include details about the associated constructs (visibility, types, etc.). These details (thereafter called parameters) determine ways a code fragment can deviate from a notion of normality. The sequence of predicates must follow the specified order of predicate types (Class, Attribute, Method, Generalization, association, etc.) to ease the comparison between predicate sequences and then reducing the computational complexity. When several predicates of the same type exist, we order them according to their parameters.



Class(C12,public);
Generalisation(C12,C9);
Attribute(C12, a254,long,static);
Attribute(C12, a54, short,static);
Method(C12, m154, void,Y, public);
Parameter(C12, m154, p47,short);
Parameter(RangeExceptionImpl,RangeExceptionImpl,message, String);
Method(C12, m129, void,Y,private);

Figure 8 Solution representation: vector (GA) to generate artificial code-smell

To generate an initial population for both GP and GA, we start by defining the maximum tree/vector length (max number of metrics/code-elements per solution). The tree/vector length is proportional to the number of metrics/code-elements to use for code-smell's detection. Sometimes, a high tree/vector length does not mean that the results are more precise. These parameters can be specified either by the user or chosen randomly. Figure 4 shows an example of a generated code-smell composed of one class, one generalization link,

two attributes, two methods, and two parameters. The parameters of each predicate contain information generated randomly describing each code element (type, visibility, etc.).

3.2.2.2 Solution Evaluation

The encoding of an individual should be formalized as a mathematical function called the "fitness function". The fitness function quantifies the quality of the proposed detection rules and generated artificial code-smells. The goal is to define efficient and simple fitness functions in order to reduce the computational cost. For our GP adaptation (upper level), we used the fitness function f_{upper} defined in the previous section to evaluate detection-rules solutions. For the GA adaptation (lower level), we used the fitness function f_{lower} defined in the previous section to evaluate generated artificial code-smells.

3.2.2.2.1 Evolutionary Operators: Selection

One of the most important steps in any evolutionary algorithm (EA) is the selection phase. There are two selection phases in EAs: (1) parent selection (also named mating pool selection) and (2) environmental selection (also named replacement). In this work, we use an elitist scheme for both selection phases with the aim to: (1) exploit good genes of fittest solutions, and (2) preserve the best solutions along the evolutionary process. The two selections schemes are described as follows. Regarding the parent selection, once the population individuals are evaluated, we select the $|P|/2$ best individuals of the population P to fulfill the mating pool, which size is equal to $|P|/2$. This allows exploiting the past experience of the EA in discovering the best chromosomes' genes. Once this step is performed, we apply genetic operators (crossover and mutation) to produce the offspring

population *Q*, which has the same size as *P* ($|P| = |Q|$). Since crossover and mutation are stochastic operators, some offspring individuals can be worse than some of *P* individuals. In order to ensure elitism, we merge both populations *P* and *Q* into *U* (with $|U| = |P| + |Q| = 2|P|$), and then the population *P* for the next generation is composed of the $|P|$ fittest individuals from *U*. By doing this, we ensure that we encourage the survival of better solutions. We can say that this environmental selection is elitist, which is a desired property in modern EAs[44]. We use this elitist operation in both upper and level-level EAs.

3.2.2.2.2 *Evolutionary Operators: Mutation*

For GP (upper-level), the mutation operator can be applied to a function node, or to a terminal node. It starts by randomly selecting a node in the tree. Then, if the selected node is a terminal (quality metric), it is replaced by another terminal (metric or another threshold value); if it is a function (AND-OR), it is replaced by a new function; and if tree mutation is to be carried out, the node and its sub-tree are replaced by a new randomly generated sub-tree.

For GA (lower-level), the mutation operator consists of randomly changing a predicate (code element) in the generated predicates.

3.2.2.2.3 Evolutionary Operators: Crossover

For GP (upper-level), two parent individuals are selected, and a sub-tree is picked on each one. Then crossover swaps the nodes and their relative sub-trees from one parent to the other. This operator must ensure the respect of the depth limits. The crossover operator can be applied with only parents having the same rule category (code-smell type to detect). Each child thus combines information from both parents. In any given generation, a variant will be the parent in at most one crossover operation.

For GA (lower-level), the crossover operator allows to create two offspring o1 and o2 from the two selected parents p_1 and p_2, as follows:

(1) A random position k is selected in the predicate sequences.

(2) The first k elements of p_1 become the first k elements of o_1. Similarly, the first k elements of p_2 become the first k elements of o_2.

(3) The remaining elements of, respectively, p_1 and p_2 are added as second parts of, respectively, o_2 and o_1 (hence having a crossing-over operation between parents).

For instance, if k = 3 and p_1 = CAMMPPP and p_2 = CMPRMPP, then o_1 = CAMRMPP and o_2 = CMPMPPP.

3.3 Validation

In order to evaluate our approach for detecting code-smells using the proposed bi-level optimization (BLOP) approach, we conducted a set of experiments based on different versions of open source systems: JFreeChart, GanttProject, ApacheAnt, Nutch, Log4J,

Lucene, Xerces-J and Rhino. Each experiment is repeated 31 times, and the obtained results are subsequently statistically analyzed with the aim to compare our bi-level proposal with a variety of existing code-smells detection approaches. In this section, we first present our research questions and then describe and discuss the obtained results.

3.3.1 Research Questions

We defined five research questions that address the applicability, performance comparison with existing code-smells detection approaches, and the scalability of our bi-level code-smells detection approach. The five research questions are as follows:

RQ1: Search validation: To validate the problem formulation of our approach, we compared our BLOP formulation with Random Search (applied to both levels). If Random Search outperforms a guided search method thus, we can conclude that our problem formulation is not adequate. Since outperforming a random search is not sufficient, the next four questions are related to performance and scalability of BLOP, and a comparison with the state-of-the-art code-smells detection approaches.

RQ2: How does BLOP perform to detect different types of code-smells? It is important to quantitatively assess the completeness and correctness of our code-smell detection approach.

RQ3.1: How do BLOP perform compared to existing search-based code-smells detection algorithms? Our proposal is the first work that treats a software engineering problem as a bi-level problem. A comparison with existing search-based code-smells detection approaches is

helpful to evaluate the benefits of the use of bi-level approach in the context of code-smell detection.

RQ3.2: How does BLOP perform compared to the existing code-smells detection approaches not based on the use of metaheuristic search? While it is very interesting to show that our proposal outperforms existing search-based code-smells detection approaches, developers will consider our approach useful, if it can outperform other existing tools that are not based on optimization techniques.

RQ4: How does our bi-level formulation scale? There is a cost in solving every lower-level optimization problem in each iteration. An evaluation of the execution time is required to discuss the ability of our approach to detect code-smells within a reasonable time-frame.

3.3.2 Software Projects Studied

In our experiments, we used a set of well-known and well-commented open-source Java projects. We applied our approach to nine open source Java projects. Table II presents the list and some relevant statistics of the software systems for our code-smell detection purpose.

Table 2 Software studied in our experiments.

Systems	*Release*	*#Classes*	*#Smells*	*KLOC*
JFreeChart	v1.0.9	521	82	170
GanttProject	v1.10.2	245	67	41
ApacheAnt	v1.5.2	1024	163	255
ApacheAnt	v1.7.0	1839	159	327
Nutch	v1.1	207	72	39
Log4J	v1.2.1	189	64	31
Lucene	v1.4.3	154	37	33
Xerces-J	V2.7.0	991	106	238
Rhino	v1.7R1	305	78	57

JFreeChart is a powerful and flexible Java library for generating charts. GanttProject is a cross-platform tool for project scheduling. ApacheAnt is a build tool and library specifically conceived for Java applications. Nutch is an open source Java implementation of a search engine. Log4j is a Java-based logging utility. Lucene is a free/open source information retrieval software library. Xerces-J is a family of software packages for parsing XML. Finally, Rhino is a JavaScript interpreter and compiler written in Java and developed for the Mozilla/Firefox browser. Table II provides some descriptive statistics about these nine programs. We selected these systems for our validation because they range from different sizes that have been actively developed over the past 10 years, and include a large number of code-smells. In addition, these systems are well studied in the literature, and their code-smells have been detected and analyzed manually.

In the nine studied open source systems, the seven code-smell types, described in Section 2.1, were identified manually. In[60], Moha et al. asked three groups of students to analyze the libraries to tag instances of specific code-smells to validate their detection technique, DECOR. For replication purposes, they provided a corpus of describing instances of different code-smells. In our previous work[55, 93], we asked eighteen graduate students and four software engineers to extend the existing corpus proposed by Moha et al. To calculate the recall, the different participants analyzed different quality metrics based on the definition of code-smells. Recently, Palomba et al.[24] proposed another corpus including several new code-smells types. The selected types of code-smells in our validation are not similar (unilateral), diversified and cover different high level design quality (maintainability) attributes such as reusability, flexibility, understandability, functionality, extendibility, and effectiveness[99].

3.3.3 Evaluation Metrics Used

To assess the accuracy of our approach, we compute two measures, *precision (PR)* and *recall (Rc)*, originally stemming from the area of information retrieval. When applying precision and recall in the context of our study, the precision denotes the fraction of correctly detected code-smells among the set of all detected code-smells. The recall indicates the fraction of correctly detected code-smells among the set of all manually identified code-smells (that is, how many code-smells are undetected). In general, the “precision” denotes the probability that a detected code-smell is correct, and the “recall” is the probability that an expected code-smell is detected. Thus, both values range between 0 and 1, whereas a higher

value is better than a lower one. We performed a 9-fold cross validation thus we removed the expected code-smells to detect in the system to evaluate from the base of code-smell examples when executing our BLOP algorithm, and then precision and recall scores are calculated automatically based on a comparison between the detected code-smells and expected ones. Thus, one project is evaluated by using the remaining systems as a base of code-smell examples to generate detection rules. We also use another measure *computational time* (*CT*) to evaluate the execution time required by our proposal to generate optimal detection rules.

3.3.4 Inferential statistical test methods used

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is performed based on 31 independent simulation runs for each problem instance, and the obtained results are statistically analyzed by using the Wilcoxon rank sum test[100] with a 99% confidence level (α = 1%). The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples to verify whether their population mean-ranks differ or not. The latter verifies the null hypothesis H_0 that the obtained results of two algorithms are samples from continuous distributions with equal medians, as against the alternative that they are not, H_1. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H_0 while it is true (type I error). A p-value that is less than or equal to α (≤ 0.01) means that we accept H_1, and we reject H_0. However, a p-value that is strictly greater than α (> 0.01) means the opposite. In

this way, we could decide whether the outperformance of BLOP over one of each of the other detection algorithms (or the opposite) is statistically significant or just a random result.

To answer the first research question RQ1, an algorithm was implemented where at each iteration, rules were randomly generated in the upper level, and artificial code-smells were randomly created. The obtained best detection rules solution was compared for statistically significant differences with BLOP using *PR, Rc* and *CT*.

To answer RQ2, we used the open source systems described in Section 4.2 and calculated precision and recall scores for the different types of code-smells. To answer RQ3.1, we compared BLOP with two existing search-based code-smells detection approaches – GP[7] and a co-evolutionary approach[98]. In[7], Kessentini et al. used genetic programming (GP) to generate detection rules from manually collected code-smell examples which correspond to the upper level of our approach (without lower level). Thus, the generation of artificial code-smells is not considered but only the coverage of manually identified examples. In [98], we proposed the use of co-evolutionary (Co-Evol) algorithms for code-smell detection, where two populations were evolved in parallel. The first population generates detection rules and the second population generates artificial code-smell examples. Both populations are executed in parallel without hierarchy. Thus, the second population solutions are independent of the solutions in the first population which is one of the main differences with our bi-level extension. We considered the three metrics *PR, Rc* and *CT* to compare bi-level with these search-based techniques based on 31 independent executions. To answer RQ3.2, we compared our results with DECOR. Moha et al.[60] started by describing code-smell

symptoms using a domain-specific-language (DSL) for their approach called DECOR. They proposed a consistent vocabulary and DSL to specify anti-patterns based on the review of existing work on design code-smells found in the literature. To describe code-smell symptoms, different notions are involved, such as class roles and structures. Symptom descriptions are later mapped to detection algorithms based on a set of rules. We compared the results of this tool with BLOP using *PR* and *Rc* metrics.

To answer the last question RQ4 we evaluated the execution time *CT* required by our BLOP proposal based on different scenarios (parameters setting) on a large scale system.

3.3.5 Parameter tuning

An often-omitted aspect in metaheuristic search is the tuning of algorithm parameters[50]. In fact, parameter setting influences significantly the performance of a search algorithm on a particular problem. For this reason, for each search algorithm and each system (cf. Table III), we performed a set of experiments using several population sizes: 10, 20, 30, 40 and 50. The stopping criterion was set to 750,000 fitness evaluations for all algorithms in order to ensure fairness of comparison. We used a high number of evaluations as a stopping criterion since our bi-level approach requires involves two levels of optimization. Each algorithm was executed 31 times with each configuration and then comparison between the configurations was performed based on precision and recall using the Wilcoxon test. Table III reports the best configuration obtained for each couple (algorithm, system). Since the replication of the bi-level experiment for 31 times is computationally expensive, the parameter setting

experiments are performed on a cluster of 30 machines. In this way, each 30 experiments are performed in parallel with a termination criterion of 750000 evaluations.

Table 3 Best population size configurations

System	Release	BLOP	CO-EVOL	GP
JFreeChart	v1.0.9	30	30	40
GanttProject	v1.10.2	30	50	30
ApacheAnt	v1.5.2	30	30	50
ApacheAnt	v1.7.0	30	30	30
Nutch	v1.1	30	40	30
Log4J	v1.2.1	30	30	50
Lucene	v1.4.3	30	40	50
Xerces-J	V2.7.0	30	40	40
Rhino	v1.7R1	30	30	30

The other parameters' values were fixed by trial and error and are as follows: (1) crossover probability = 0.8; mutation probability = 0.5 where the probability of gene modification is 0.3; stopping criterion = 750000 fitness evaluations. For our bi-level approach, both lower-level and upper-level EAs are run each with a population of 30 individuals and 50 generations. The following reasons can explain the use of reduced population size at both levels. According to our formulation, detection rules are evaluated at the upper level based not only on its performance with respect to the upper fitness function but also on its performance on detecting associated generated code-smells by the lower level. In this way, the lower level helps the upper one in (1) discovering uninteresting upper search

directions that should be ignored, and (2) favoring interesting ones; thereby we reduce the number of required evaluations at the upper level. This fact explains the reduced population size (30 individuals) at the upper level. For the lower level, we also used a reduced population size (30 individuals) since our goal is to have an idea about the performance of the detection rule at the lower level. We note that recent studies are based on the use of local search in order to *predict the* behavior of upper solutions at a lower level with an accepted computational cost (cf.[44]).

It should be noted that the lower-level routine is not called for all upper-level population members. To control, the high computational cost of our bi-level approach, only *nbs*% of the best upper-level population members are allowed to call the lower-level optimization algorithm. Based on a parametric study, the value of 10% for *nbs* is found to be adequate empirically in our experiments. The *nbs* parametric study will be discussed later. For our experiment, we generated up to 125 artificial code-smells from deviation with JHotDraw (about a quarter of the number of examples). JHotdraw was chosen as an example of reference code because it contains very few known code-smells. In fact, previous work[98] could not find any Blob in JHotdraw. In our experiments, we used all the classes of JHotdraw as our example set of well-designed code.

3.3.6 Results

This section describes and discusses the results obtained for the different research questions of Section 4.1.

3.3.6.1 Results for RQ1

Concerning RQ1, Table IV confirms that BLOP is better than random search based on the two metrics *PR* and *Rc* on all the 9 systems. The Wilcoxon rank sum test showed that in 31 runs BLOP results were significantly better than random search.

Table 4 The significantly best algorithm among random search, BLOP, GP and Co-Evol over 31 independent runs. "No. Sign." Means no method is significantly better than another.

System	Release	Precision	Recall
JFreeChart	v1.0.9	BLOP	BLOP
GanttProject	v1.10.2	BLOP	BLOP
ApacheAnt	v1.5.2	BLOP	BLOP
ApacheAnt	v1.7.0	BLOP	BLOP
Nutch	v1.1	BLOP	BLOP
Log4J	v1.2.1	No. Sign.	No. Sign.
Lucene	v1.4.3	No. Sign.	No. Sign.
Xerces-J	V2.7.0	BLOP	BLOP
Rhino	v1.7R1	BLOP	BLOP

Table 5 . Median PR and Rc values on 31 runs for BLOP, random search (RS), GP [Kessentini et al. 2011] and Co-Evol [Boussaa et al. 2013]. The results were statistically significant on 31 independent runs using the Wilcoxon rank sum test with a 99% confidence le

System	*PR-BLOP*	*PR-GP*	*PR-Co-Evol*	*PR-RS*	*Rc-BLOP*	*Rc-GP*	*Rc-Co-Evol*	*Rc-RS*
JFreeChart	89% (77/86)	78% (71/92)	84% (74/89)	26% (34/129)	93% (77/82)	86% (71/82)	90% (74/82)	41% (34/82)
GanttProject	88% (62/71)	80% (57/73)	82% (58/71)	28% (29/106)	89% (62/67)	83% (57/67)	85% (58/67)	43% (29/67)
ApacheAnt v1.5.2	90% (152/169)	84% (146/174)	86% (148/171)	26% (51/189)	93% (152/163)	89% (146/163)	90% (148/163)	31% (51/163)
ApacheAnt v 1.7.0	91% (149/164)	82% (142/173)	85% (144/169)	28% (54/184)	94% (149/159)	90% (142/159)	91% (144/159)	33% (54/159)
Nutch	89% (67/76)	73% (64/88)	76% (65/86)	34% (37/131)	92% (67/72)	89% (64/72)	90% (65/72)	51% (37/72)
Log4J	89% (59/67)	71% (52/74)	77% (54/71)	32% (34/127)	91% (59/64)	81% (52/64)	85% (54/64)	53% (34/64)
Lucene	91% (35/39)	70% (31/42)	79% (33/42)	12% (11/88)	95% (35/37)	84% (31/37)	89% (33/37)	29% (11/37)
Xerces-J	91% (101/111)	75% (94/126)	80% (96/119)	17% (31/179)	95% (101/106)	88% (94/106)	91% (96/106)	29% (31/106)
Rhino	90% (75/84)	74% (69/93)	79% (71/89)	14% (23/167)	95% (75/78)	87% (69/78)	91% (71/78)	29% (23/78)

Table V also describes the outperformance of BLOP against random search. The average precision and recall values of random search on the nine systems are lower than 25%. This can be explained by the huge search space to explore at both levels to generate detection rules and artificial code-smells. We conclude that there is empirical evidence that our bi-level formulation surpasses the performance of random search thus our formulation is adequate (this answers RQ1). Table V. Median PR and Rc values on 31 runs for BLOP, random search (RS), GP[7] and Co-Evol[98]. The results were statistically significant on 31 independent runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).

3.3.6.2 Results for RQ2

In this section, we evaluate the performance of our BLOP adaptation on the detection of seven different types of code-smells. Table V summarizes our findings. The expected code-smells were detected with an average of more than 90% of precision and recall on the nine open source systems. The highest precision was found in JDI-Ford, Xerces-J and Lucene where 91% of code-smells were detected. The lowest precision was found in GanttProject with 88% of detected code-smells. This can confirm that the list of returned code-smells did not contain high number of false positive thus developer will not waste a lot of their time for manual inspections. We found similar facts when analyzing the recall scores of BLOP on the different system where an average of more than 90% of expected code-smells was detected. The highest and lowest recall scores were respectively 95% (Rhino) and 89% (GanttProject). An interesting observation that the performance of bi-level in terms of PR and Rc is

independent of the size and number of code-smells in the system to analyze. The PR and Rc scores of ApacheAnt are among the highest ones with more than 90% that are better than the detection results of a smaller system such as GanttProject. A more qualitative evaluation is presented in Figure 5 illustrating the box plots obtained for the PR and Rc metrics on three different projects GanttProject (small), Xerces-J (medium) and Ant-Apache (large). We see that for almost all problems the distributions of the PR and Rc values for BLOP are the best ones.

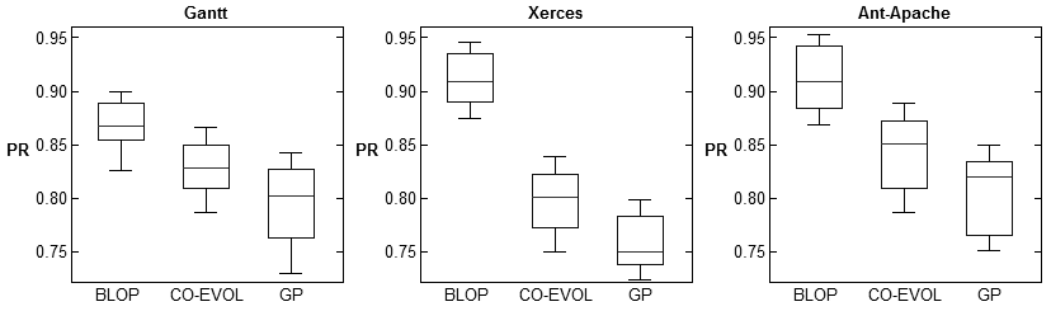


Figure 9 Box plots on three different systems (Gantt: small, Xerces: medium, Ant-Apache 1.7.0: large) of precision values

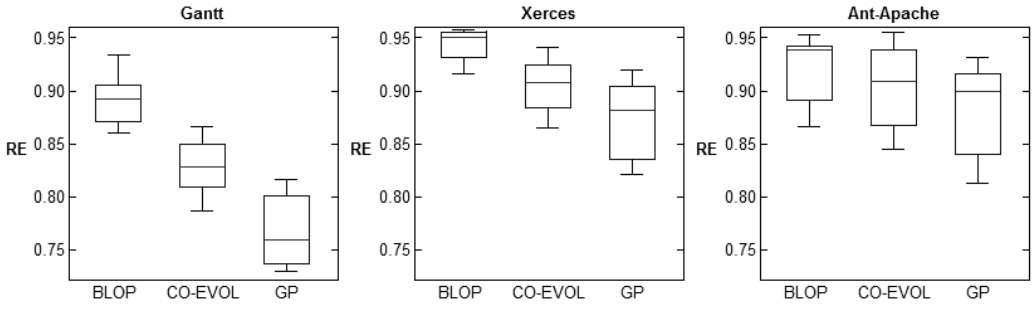


Figure 10 Box plots on three different systems (Gantt: small, Xerces: medium, Ant-Apache 1.7.0: large) recall values.

We noticed that our technique does not have a bias towards the detection of specific code-smell types. As described in Figure 6, in all systems, we had an almost equal distribution of each code-smell types. Having a relatively good distribution of code-smells is useful for a

quality engineer. Overall, all the seven code-smells types are detected with good precision and recall scores in the different systems (more than 80%).

This ability to identify different types of code-smell underlines a key strength to our approach. Most other existing tools and techniques rely heavily on the notion of size to detect code-smells. This is reasonable considering that some code-smells like the Blob are associated with the notion of size. For code-smells like FDs, however, the notion of size is less important, and this makes this type of anomaly hard to detect using structural information.

To conclude, our BLOP approach detects well all the seven types of considered code-smells (RQ2).

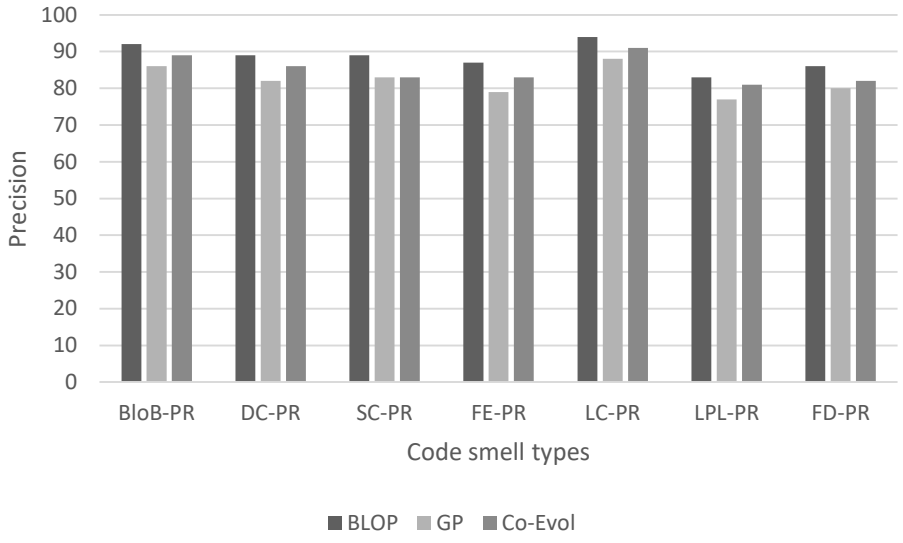


Figure 11 Median PR scores for every code-smell's type over 31 runs on the different 9 open source systems.

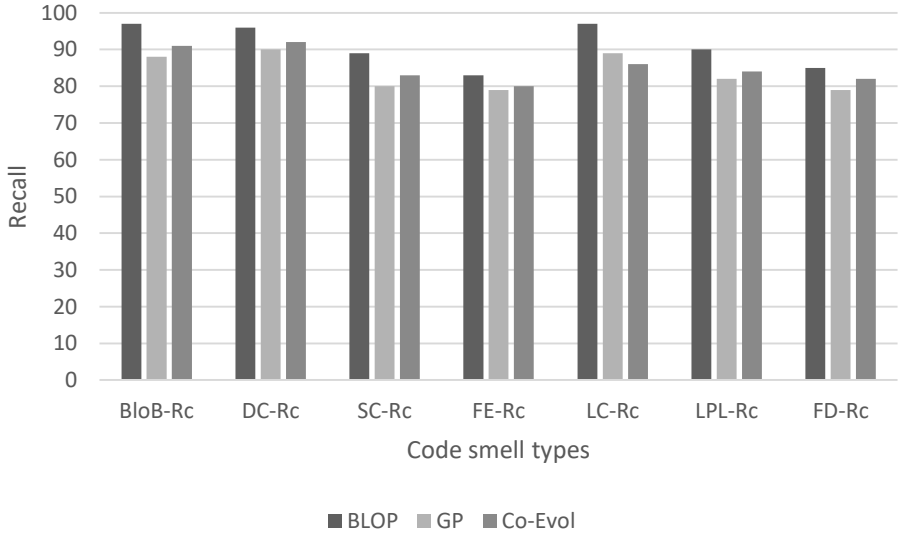


Figure 12 Median Rc (b) scores for every code-smell's type over 31 runs on the different 9 open source systems.

3.3.6.3 Results for RQ3

In this section, we compare our BLOP adaptation to the current; state-of-the-art code smells detection approaches. To answer RQ3.1, we compared BLOP to two other existing search-based techniques: GP[7] and Co-Evol[98]. Table V shows the overview of the results of the significance tests comparison between all these algorithms. It is clear that BLOP outperforms GP and Co-Evol in 100% of the cases in terms of precision (PR) and recall (RE). However, as it will be discussed in RQ4, the execution time (CT) of our BLOP algorithm is much higher than GP and Co-Evol. In addition, the improvements of PR and Rc scores are significant using BLOP comparing to GP and Co-Evol. A more qualitative evaluation is presented in Figure 5 illustrating the box plots obtained for the PR and Rc metrics on three different projects GanttProject (small), Xerces-J (medium) and Ant-Apache (large). It is clear from the box plots presented in Figure 5 that the outperformance of BLOP comparing to GP and Co-Evol in all the three different systems based on the statistical

analysis of 31 independent runs. For GP, this can be explained by the fact that the use of manually identified code-smell examples is not enough to cover all the possible bad-practice behaviors since it is fastidious task to collect them (most of the code-smells are not well-documented, unlike bugs report). However, our BLOP algorithm can generate artificial code-smells based on a deviation with good-practices in addition to the coverage of code-smell examples. For Co-Evol, the two populations are executed in parallel and the problem is that there is no dependency between both populations (unlike BLOP that creates a hierarchy between two levels) thus one population can converge before the second one.

The statistical tests are based on multiple pairwise comparisons using the Wilcoxon test. Thus, we have to adjust the p-values. To achieve this task, we used Holm method that is reported to be more accurate than the Bonferroni one[101]. Table VI presents these adjusted p-values confirming that the results are statistically significant with a 99% confidence level ($\alpha = 1\%$).

Table 6 Adjusted p-values of comparisons related to table 4

	BLOP	GP	Co-Evol
JFreeChart	0.00353 (GP) 0.00463 (Co-Evol) 0.00157 (RS)	0.00379 (Co-Evol) 0.00272 (RS)	0.00377 (RS)
GanttProject	0.00116 (GP) 0.00249 (Co-Evol) 0.00039 (RS)	0.00296 (Co-Evol) 0.00128 (RS)	0.00138 (RS)
ApacheAnt v1.5.2	0.00273 (GP) 0.00479 (Co-Evol) 0.00282 (RS)	0.00386 (Co-Evol) 0.00053 (RS)	0.00240 (RS)
ApacheAnt v 1.7.0	0.00179 (GP) 0.00485 (Co-Evol) 0.00079 (RS)	0.00216 (Co-Evol) 0.00138 (RS)	0.00228 (RS)
Nutch	0.00226 (GP) 0.00391 (Co-Evol) 0.00111 (RS)	0.00431 (Co-Evol) 0.00294 (RS)	0.00321 (RS)
Log4J	0.00243 (GP) 0.00405 (Co-Evol) 0.00071 (RS)	0.00412 (Co-Evol) 0.00147 (RS)	0.00195 (RS)
Lucene	0.00211 (GP) 0.00458 (Co-Evol) 0.00196 (RS)	0.00259 (Co-Evol) 0.00175 (RS)	0.00267 (RS)
Xerces-J	0.00480 (GP) 0.00328 (Co-Evol) 0.00017 (RS)	0.00217 (Co-Evol) 0.00119 (RS)	0.00226 (RS)
Rhino	0.00325 (GP) 0.00467 (Co-Evol) 0.00239 (RS)	0.00391 (Co-Evol) 0.00183 (RS)	0.00249 (RS)

We found that the main reason explaining the outperformance of BLOP against Co-Evol is the diversity/quality of the generated artificial code-smells. In fact, the lower level of our BLOP formulation generates artificial code-smell examples for every good solution (detection rules) in the upper level then these examples are used to evaluate the solutions in the upper level. Thus, the generated artificial code-smells examples depend on the associated solution (rules) in the upper level. However, both populations, in Co-Evol, are executed in

parallel and independently (without the hierarchy of BLOP). Thus, the generated artificial code-smells by the second population can be evaluated, at every iteration, by the best solution (detection rule) of the first population that is not “mature” enough. In fact, one population in Co-Evol can converge before the other however this is addressed by BLOP because the two levels are executed in sequence after number of iterations. We executed the best solution (code-smell's detection rules) of BLOP to detect code-smells on the best set of artificial code-smells generated Co-Evol and vice-versa. As described in Figure 7, it is clear that the rules generated by BLOP can detect most of the artificial code-smells examples generated by Co-Evol with a precision of more than 90% however Co-Evol detects only around 70% of code-smells generated by the lower level of BLOP. The quality of generated rules depends heavily on the diversity of the artificial code-smells. Thus, the main reason explaining the outperformance of BLOP is the hierarchy that exists between the two levels allowing the lower level to generate intelligently the artificial code-smells that can improve the quality of detection rules.

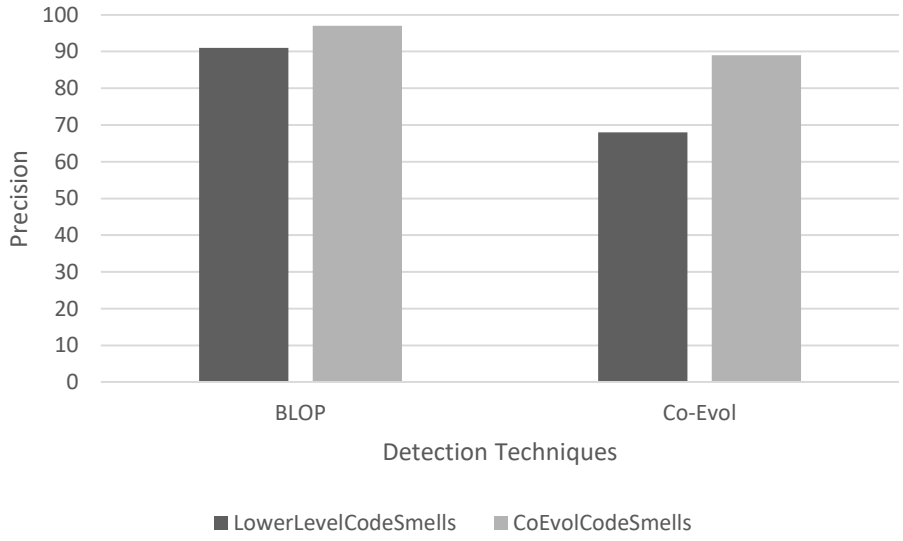


Figure 13 The median precision score of detected artificial code-smells by BLOP and Co-Evol

We analyzed the results of our comparison between the different search techniques based on a final target fitness function value and not a maximum number of iterations. Due to the space limitation, we present in Figure 8 the results of the comparison based on precision and recall only on Xerces-J but similar facts were found on the remaining systems. The four different targeted fitness functions value (0.7, 0.75, 0.8, 0.85) confirms the outperformance of BLOP comparing to the other search techniques in terms of precision and recall. An interesting observation is the fact that the fitness function values are correlated with the precision and recall scores. An improvement of the fitness function value leads to better precision and recall scores. This is can be a good indication about our fitness functions is well formulated and adapted to the code-smell's detection problem. We also noticed that all the targeted fitness functions values are reached by all the algorithms with a lower number of evaluations than 750000 which the termination criterion selected in our experiments.

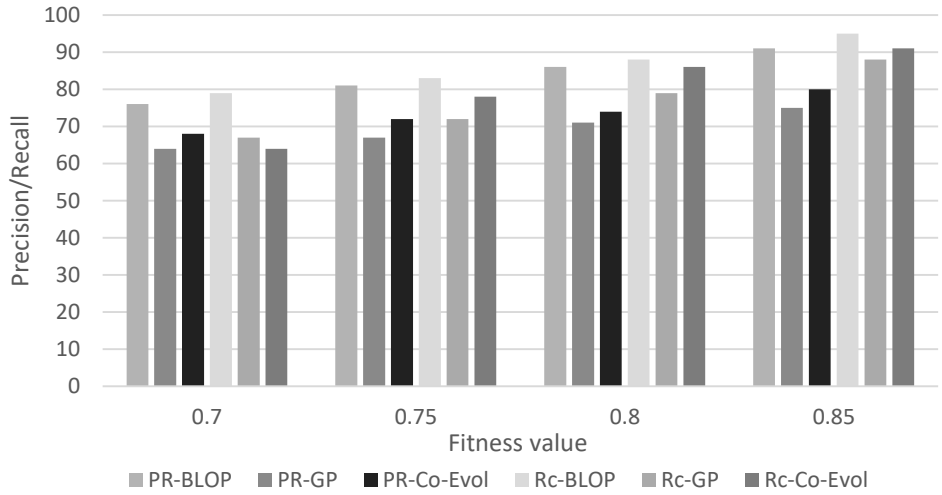


Figure 14 The median precision and recall scores of detected code-smells by BLOP, GP and Co-Evol on Xerces-J based on a target final fitness function values

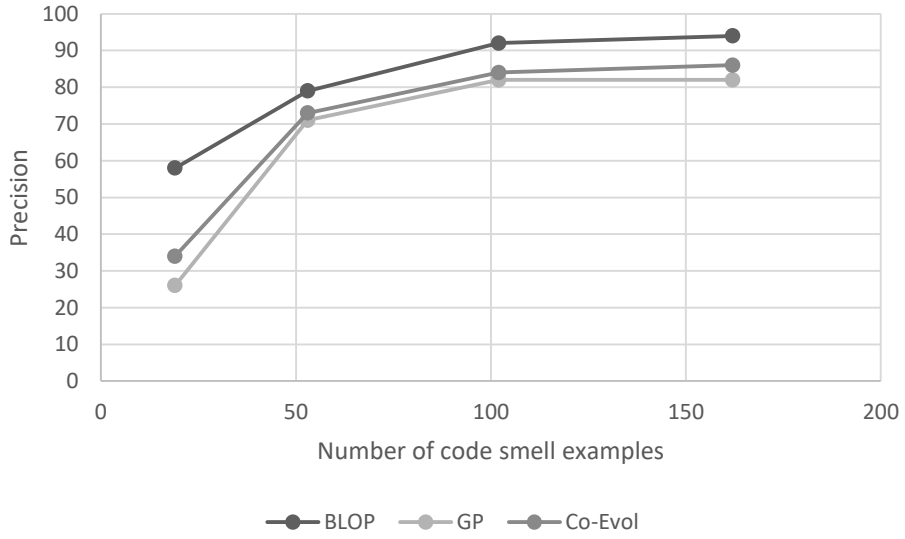


Figure 15 The impact of the number of code-smell examples on the quality of the results (PR on Xerces-J).

One of the advantages of using our BLOP adaptation is that the developers do not need to provide a huge set of code-smell examples to generate the detection rules. In fact, the lower-

level optimization can generate examples of code-smells that are used to evaluate the detection rules at the upper level. Figure 9 shows that BLOP requires a low number of manually identified code-smells to provide good detection rules with reasonable precision and recall scores. GP and Co-Evol require a higher number of code-smell examples than BLOP to generate good code-smells detection rules. In addition, the reliability of the proposed BLOP approach requires an example set of good code and code-smell examples. In our study, we showed that by using JHotdraw directly, without any adaptation, the BLOP method can be used out of the box, and this will produce good detection results for the detection of code-smells for all the studied systems. In an industrial setting, we could expect a company to start with JHotDraw, and gradually transform its set of good code examples to include context-specific data. This might be essential if we consider that different languages and software infrastructures have different best/worst practices.

In conclusion, we answer RQ3.1 by concluding that the results in our experiments confirm that our proposed BLOP is adequate, and it outperforms two existing search-based code-smells detection work – a GP[7] and a co-evolutionary GA[98].

Since it is not sufficient to compare our proposal with only search-based work, we compared the performance of BLOP with DECOR[60]. DECOR was mainly evaluated based on three types of code-smells using metrics-based rules that are identified manually: Blob, functional decomposition and spaghetti code. The results of the execution of DECOR are only available for the following systems: GanttProject, Nutch, Log4J, Lucene and Xerces-J. Figure 10 summarizes the results of the precision and recall obtained on the above-mentioned

5 systems. The recall for DECOR in all the systems is 100% signifying that it is better than BLOP; however the precision scores are lower than our proposal on all systems. In fact, the higher recall achieved by DECOR can be easily explained by the use of relatively permissive constraints. For example, the average precision to detect functional decompositions using DECOR is lower than 25% however we can detect this type of code-smells with more than 80% of precision. The same observation for the spaghetti-code (SC), BLOP can detect SC with more than 85% in terms of precision however DECOR detected the same type of code-smell with a precision lower than 65%. This can be explained by the high calibration effort required to define manually the detection rules in DECOR for some specific code-smell types and the ambiguities related to the selection of the best metrics set. The recall of BLOP is lower than DECOR, on average, but it is an acceptable recall average which is higher than 80%. To conclude, our BLOP adaption also outperforms, on average, an existing approach not based on meta-heuristic search (RQ3.2).

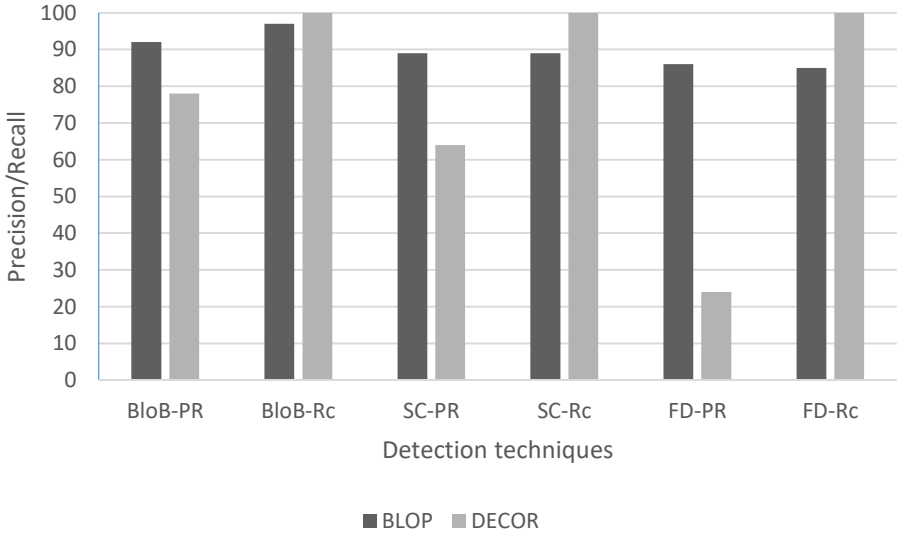


Figure 16 The median precision and recall scores of BLOP and DECOR obtained on GanttProject, Nutch, Log4J, Lucene and Xerces-J based on three code-smell types (Blob, SC and FD).

3.3.6.4 Results for RQ4

Since our proposal is based on bi-level optimization, it is important to evaluate the execution time (*CT*). It is evident that BLOP requires higher execution time than GP, Co-Evol and DECOR since BLOP has an optimization algorithm to be executed at the lower level. To reduce the computational complexity of our BLOP adaptation, we selected only best solutions (*nbs*%) at the upper level to update their fitness evaluations based on the coverage of artificial code-smells that are generated by the optimization algorithms executed at the lower level for every selected solution. All the search-based algorithms under comparison were executed on machines with Intel Xeon 3 GHz processors and 4 GB RAM. We recall that all algorithms were run for 750000 evaluations. This allows us to make a fair comparison between CPU times. Overall, GP and Co-Evol algorithms were faster than BLOP. In fact, the average execution time for BLOP, GP and Co-Evol were respectively 6.2,

1.4 and 2.1 hours. However, the execution for BLOP is reasonable because the algorithm is executed only once then the generated rules will be used to detect code-smells. There is no need to execute BLOP again except in the case that the base of examples will be updated with a high number of new code-smell examples.

An important parameter that reduced the execution time of our BLOP adaptation is the number of selected good solutions at the upper level. Figure 11 shows that the performance of our approach improves as we increase the percentage of best solutions selected from the upper level at each iteration. However, the results become stable after 10% (percentage of selected solutions from the upper level population). For this reason, we considered this threshold in our experiments that represent a good trade-off between the quality of detection solutions and the execution time. As described in Figure 11, the PR and Rc scores become almost stable after the 10% threshold value and the execution time increases dramatically since a high number of optimization algorithms are executed at the lower level. The evaluation was performed on JFreeChart v1.0.9 but similar facts were found on the remaining systems.

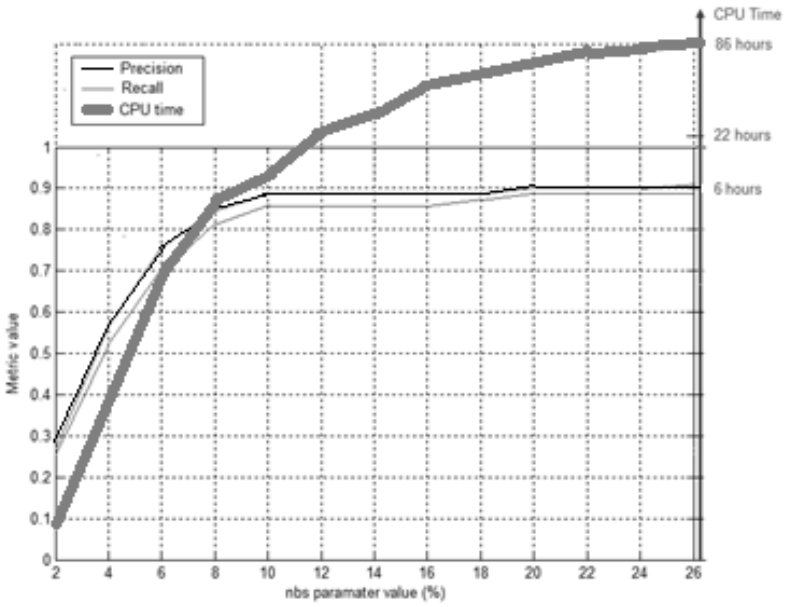


Figure 17 The impact of the number of selected solutions at upper level on the quality of the results (PR, Rc and CT) using JFreeChart v1.0.9.

Figure 12 shows the number of evaluations required to generate good code-smells detection rules. We used the *f-measure* metric defined as the harmonic mean of precision and recall to evaluate the quality of the best solution at each iteration for each algorithm (BLOP, Co-Evol and GP). We considered an *f-measure* value higher than 0.7 as an indication of an acceptable detection rules solution based on our corpus. We evaluated which algorithm can reach faster that threshold value of *f-measure (0.7)*. We selected a threshold value of 0.7 since it represents a good balance between precision and recall that can lead to acceptable detection solutions. We found that our bi-level adaptation required a fewer number of evaluations than Co-Evol and GP to generate good code-smells detection solutions. In fact, after around 325000 evaluations BLOP generated detection rules that have 0.7 as *f-measure* value on Xerces-J. Co-Evol requires at least more than 480000 evaluations to reach similar

solution quality, and GP needs more than 560000 evaluations to generate similar detection solution. Thus, we can conclude that the lower level helped the upper level to generate quickly good quality of detection solutions by producing in an intelligent manner efficient artificial code-smells. Although the fact that BLOP needs higher execution time than Co-Evol and GP as described in Figure 11, it is clear from Figure 12 that the good solutions provided by a single-level approach can be reached quickly by our bi-level adaptation.

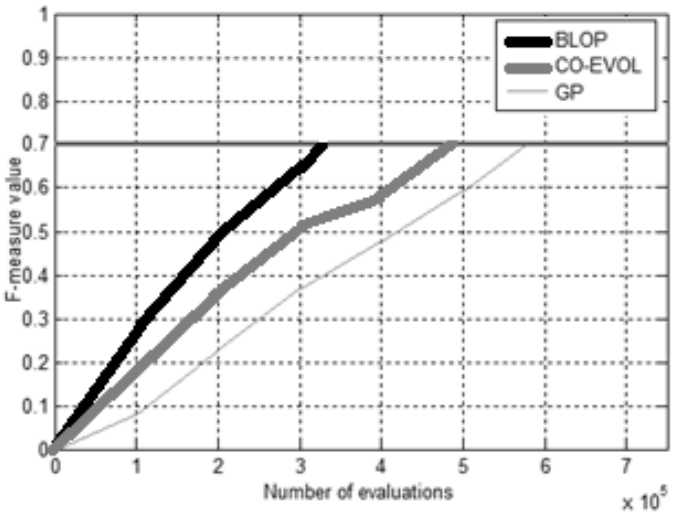


Figure 18 The number of evaluations required by the different algorithms (BLOP, Co-Evol and GP) to reach acceptable results (f-measure=0.7) using Xerces-J v 2.7.0.

To further evaluate the scalability of the performance of bi-level evolutionary algorithms for systems of increasing size, we executed our bi-level tool on Eclipse without assessing the precision and recall scores. Eclipse is an open source integrated development environment (IDE) written in Java and widely used to develop applications. We considered three versions of Eclipse that contains more than 3.5 MLOCs. Figure 13 describes the execution time of our bi-level approach on 7 different versions of Eclipse. We believe that an execution time of 7 hours is acceptable and reasonable since the developers will not use our tool in their daily

activities, but they just need to execute it once to extract the rules. A new execution of the bi-level algorithm is required when major updates are performed on the base of examples used by the upper level.

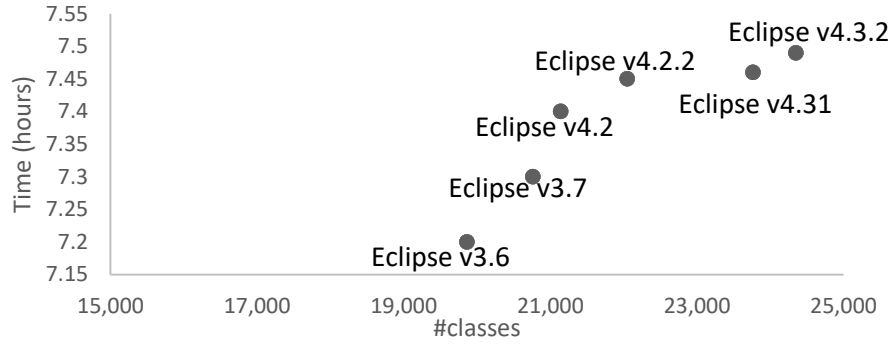


Figure 19 Scalability of our bi-level approach for code-smell's detection on three different versions of Eclipse.

3.4 Industrial Case Study and Relevance of the Detected Code-Smells

The goal of this study is to evaluate the usefulness and the effectiveness of our code-smells detection tool in practice. We conducted a non-subjective evaluation with potential developers who can use our tool related to the relevance of our approach for software engineers.

We performed a small industrial case study, described in Table VII, based on one industrial project JDI-Ford. It is a Java-based software system that helps, our industrial partner, the Ford Motor Company, analyzes useful information from the past sales of dealerships data and suggests which vehicles to order for their dealer inventories in the future. This system is the main key software application used by the Ford Motor Company to improve their vehicle sales by selecting the right vehicle configuration to the expectations of

customers. Several versions of JDI were proposed by software engineers at Ford during the past 10 years. Due to the importance of the application and the high number of updates performed during a period of 10 years, it is critical to make sure that all the JDI releases are within a good quality to reduce the time required by developers to introduce new features in the future. The software engineers from Ford evaluated the JDI system to manually find code-smells based on their knowledge of the system since they are some of the original developers. We considered those that are detected by the majority of the software engineers to calculate the precision and recall. Our study focused on the usefulness of the detected code-smells and the performance of our detection technique in an industrial setting. We also evaluated the relevance of some of the detected code-smells on the different open source systems described in the previous section.

In this section, we will answer to the following question:

How our bi-level code-smells detection BLOP approach and the detected code-smells can be useful for software engineers?

Table 7 Software studied in our experiments.

Systems	*Release*	*#Classes*	*#Smells*	*KLOC*
JDI-Ford	v5.8	638	88	247

We describe, first, in this section the subjects participated in our study. Second, we give details about the questionnaire, instructions and the conducted pilot study. Finally, we describe and discuss the obtained results.

3.4.1 Subjects

Our study involved 7 subjects from the University of Michigan and 8 software engineers from Ford Motor Company. Subjects include 3 master students in Software Engineering, 3 Ph.D. students in Software Engineering, 1 faculty member in Software Engineering, 6 junior software developers and 2 senior projects manager. 6 subjects are female, and 9 are male. All the 15 subjects are familiar with Java development, software maintenance activities including refactoring. The experience of these subjects on Java programming ranged from 2 to 16 years. All the graduate students have an industrial experience of at least 2 years with large-scale object-oriented systems. The 8 software engineers from Ford evaluated the code-smell's detection results only on the JDI-Ford system. They were selected, as part of a project funded by Ford, based on having similar development skills, their motivations to participate in the project and their availability. They are part of the original developers' team of the JDI system.

3.4.2 Questionnaire, Instructions and Pilot Study

Subjects were first asked to fill out a pre-study questionnaire containing five questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, their familiarity with code smells and software refactoring.

We divided the subjects into 4 groups to evaluate the relevance of some detected code-smells according to the number of studied systems, the number of detected code-smells to evaluate and the results of the pre-study questionnaire. Due to the high number of code-

smells to be evaluated, we pick at random a sub-set of the detected code-smells to evaluate their relevance for software engineers as described in Table VIII. In Table 8, we summarize how we divided subjects into 4 groups. All the participants from Ford (Group D) evaluated the relevance of the detected code-smells only on the JDI-Ford system. They were also asked to find the other code-smells in JDI not detected by our BLOP technique and evaluate the correctness of detected ones.

Each group of subjects, who accepted an invitation to participate to the study, received a questionnaire, a manuscript guide that helps to fill the questionnaire, and the source code of the studied systems, in order to evaluate the relevance of the suggested code-smells to fix. The questionnaire is organized in an excel file with hyperlinks to visualize the source code of the affected code elements easily. The study questionnaire is composed by three main sections and a specific fourth section only for Group D (Ford's developers).

The first part of the questionnaire includes questions to evaluate the relevance of some detected code-smells using the following scale: 1. Not at all relevant; 2. Slightly relevant; 3. Moderately relevant; and 4.Extremely relevant. If a detected code-smell is considered relevant then this is mean that the developer considers that it is important to fix it.

The second part of the questionnaire includes questions for those code-smells that are considered at least "moderately relevant", we asked the subjects to specify their usefulness to perform some maintenance activities: 1. Refactoring guidance; 2. Quality assurance; 3. Bug prediction; 4. Effort prediction; and 5. Code inspection.

In the third part of the questionnaire, we asked our subjects to fix some of the detected code-smells by suggesting and applying some refactorings[101].

In the last part dedicated to only Group D (developers from Ford), we asked them to explore the JDI source code, analyze the metrics value of the system and use their knowledge of the existing implementation (since they are the original developers) in order to identify the remaining code-smells not detected by our BLOP technique. In addition, they evaluated the correctness of detected ones by BLOP. We considered the majority of votes (more than 4 votes) to evaluate the correctness of detected code-smells.

The questionnaire is completed anonymously thus ensuring confidentiality and this study were approved by the IRB at the University of Michigan: “Research involving the collection or study of existing data, documents, records, pathological specimens, or diagnostic specimens, if these sources are publicly available or if the information is recorded by the investigator in such a manner that participants cannot be identified, directly or through identifiers linked to the participants”.

During the entire process, subjects were encouraged to think aloud and to share their opinions, issues, detailed explanations and ideas with the organizers of the study (one graduate student and one faculty from the University of Michigan) and not only answering the questions.

A brief tutorial session was organized for every participant around code-smells and refactoring to make sure that all of them have a minimum background to participate in the study. The instructions indicate also that the developers need to inspect the source code to

evaluate the detected code-smells and their relevance and not by evaluating the quality metric values. In addition, all the developers performed the experiments in a similar environment: similar configuration of the computers, tools (Eclipse, Excel, etc.) and facilitators of the study. Because some support was needed for the installation of our Eclipse plug-in and the other detection techniques considered in our experiments, we added a short description of this instruction for the participants. These sessions were also recorded as audio and the average time required to finish all the questions was 5 hours. The average time to answer the first three parts of the questionnaire is 3 hours, but the participants from Ford spent a considerable time to find the code-smells not detected by our BLOP technique.

Prior to the actual experiment we did a pilot run of the entire experiment with two subjects, on average performing student and one software engineer from Ford. We performed this pilot study to verify whether the assignments were clear and if our estimation of the required time to finalize the experiments evaluation were realistic thus all the assignments could be completed in an afternoon session by the subjects. The pilot study pointed out that the assignments and the questions in the questionnaire form were clear and relevant, and that they could be executed as offered by the subjects of the pilot study within 4 hours. The pilot study also pointed out that the description of code-smells and the examples were clear and sufficient to understand the different types of code-smells considered in our experiments. However, the pilot study showed that subjects have problems to efficiently evaluate the usefulness of the identified code-smells especially for the open source systems since they are not the original developers and could not understand all the design decisions. For this reason,

we extended our experiments to include the industrial project from Ford and some of their original developers.

Table 8 Survey organization to study the relevance of some detected code-smells.

Subject groups	Systems		Selected Code-smells
Group A	JFreeChart	v1.0.9	*Blob*: 4 *Data Class*: 8 *Spaghetti Code*: 4 *Feature Envy*: 6 *Lazy Class*: 4 *Long Parameter List*: 6 *Functional Decomposition*: 3
	GanttProject	v1.10.2	*Blob*: 4 *Data Class*: 3 *Spaghetti Code*: 4 *Feature Envy*: 2 *Lazy Class*: 2 *Long Parameter List*: 3 *Functional Decomposition*: 4
	ApacheAnt	v1.5.2	*Blob*: 5 *Data Class*: 2 *Spaghetti Code*: 3 *Feature Envy*: 3 *Lazy Class*: 2 *Long Parameter List*: 2 *Functional Decomposition*: 2
Group B	ApacheAnt	v1.7.0	Blob: 5 Data Class: 2 Spaghetti Code: 3 Feature Envy: 3 Lazy Class: 2 Long Parameter List: 2 Functional Decomposition: 3

Group C	Nutch v1.1		Blob: 4 Data Class: 3 Spaghetti Code: 3 Feature Envy: 2 Lazy Class: 4 Long Parameter List: 3 Functional Decomposition: 6
	Log4J v1.2.1		Blob: 2 Data Class: 3 Spaghetti Code: 2 Feature Envy: 4 Lazy Class: 1 Long Parameter List: 3 Functional Decomposition: 3
	Lucene	v1.4.3	Blob: 2 Data Class: 2 Spaghetti Code: 3 Feature Envy: 2 Lazy Class: 2 Long Parameter List: 3 Functional Decomposition: 2
	Xerces-J	V2.7.0	Blob: 3 Data Class: 2 Spaghetti Code: 4 Feature Envy: 4 Lazy Class: 2 Long Parameter List: 4 Functional Decomposition: 4
	Rhino v1.7R1		Blob: 2 Data Class: 2 Spaghetti Code: 2 Feature Envy: 4 Lazy Class: 4 Long Parameter List: 2

Group D	JDI-Ford v5.8	Functional Decomposition: 4 Blob: 6 Data Class: 8 Spaghetti Code: 8 Feature Envy: 8 Lazy Class: 12 Long Parameter List: 12 Functional Decomposition: 12

3.4.3 Results of the Industrial Case Study and Relevance of Detected Code-Smells

For the Ford system, the original developers conducted the study (group D). Hence, they are almost sure whether a code fragment is affected by code-smell or not, whether it should be refactored, or whether, although a tool says this is a smell, there are good reasons for the design and implementation choices made. Unfortunately, this is not possible for the other subjects and open source systems, who have evaluated code not familiar to them. For this reason, we describe the results of the evaluation in two separate sections.

3.4.3.1 Industrial Case Study : Detection Results and Relevance of Detected Code Smells

In this section, we evaluate the performance of our BLOP detection technique on the detection of seven different types of code-smells in an industrial setting and compare it with some existing code smells detection techniques. As described in Table IX, the expected code-smells were detected with more than 90% of precision on the JDI-Ford system that

corresponds to the highest precision comparing to random search, GP and Co-Evol. The lowest precision was found by the random search of only 18%. The precision of Co-Evol is higher than Co-Evol and random search but lower than BLOP on the JDI industrial system. This confirms the importance and efficiency of the lower level considered in our BLOP adaptation to improve the quality of detection solutions generated by the upper level. Similar facts were found when analyzing the recall scores of BLOP on the JDI-Ford where 86% of expected code-smells was detected. The recall score of BLOP confirms its outperformance compared to random search, GP and Co-Evol.

Table X confirms that the comparison results of the different detection techniques on JDI-Ford are statistically significant with a 99% confidence level ($\alpha = 1\%$)

Table 9 Manual validation of the detected code-smells on JDI-Ford.

System	*PR-BLOP*	*PR-GP*	*PR-Co-Evol*	*PR-RS*	*Rc-BLOP*	*Rc-GP*	*Rc-Co-Evol*	*Rc-RS*
JDI-Ford v5.8	92% (76/82)	67% (63/94)	78% (69/88)	18% (27/148)	86% (76/88)	72% (63/88)	78% (69/88)	31% (27/88)

Table 10 Adjusted p-values of comparisons related to table IX.

	BLOP	GP	Co-Evol
JDI-Ford v5.8	0.00327 (GP) 0.00167 (Co-Evol) 0.00112 (RS)	0.00398 (Co-Evol) 0.00213 (RS)	0.00187 (RS)

To better investigate the relevance of the detected code-smells for software engineers, we asked group D to fix some of the detected code-smells (described in Table XIII) in JDI-Ford by manually suggesting and applying some refactorings [Holm et al. 1979] on JDI-Ford. Then, we used the QMOOD (Quality Model for Object-Oriented Design) model [99] to estimate the effect of the fixed code-smells on quality attributes. QMOOD has the advantage that define six high level design quality attributes (reusability, flexibility, understandability, functionality, extendibility, and effectiveness) that can be calculated using 11 lower level design metrics. In our study we consider the following quality attributes:

- Reusability: The degree to which a software module or other work product can be used in more than one computer program or software system.
- Flexibility: The ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.
- Understandability: The properties of designs that enable it to be easily learned and comprehended. This directly relates to the complexity of design structure.
- Effectiveness: The degree to which the design can achieve the desired functionality and behavior using OO design concepts and techniques.

Table 11 QMOOD quality factors [Bansiya et al. 2002]

Quality attribute	Quality Index Calculation
Reusability	= -0.25 * DCC + 0.25 * CAM + 0.5 * CIS + 0.5 * DSC
Flexibility	= 0.25 * DAM - 0.25 * DCC + 0.5 * MOA +0.5 * NOP
Understandability	= -0.33 * ANA + 0.33 * DAM - 0.33 * DCC + 0.33 * = CAM -0.33 * NOP + 0.33 * NOM - 0.33 * DSC
Effectiveness	= 0.2 *ANA + 0.2 *DAM + 0.2*MOA + 0.2 * MFA + 0.2 *NOP

Table XI and Table XII summarize the QMOOD formulation of these quality attributes [99].

Table 12 QMOOD metrics for design properties [Bansiya et al. 2002]

Design Property	Metric	Description
Design size	DSC	Design size in classes
Complexity	NOM	Number of methods
Coupling	DCC	Direct class coupling
Polymorphism	NOP	Number of polymorphic methods
Hierarchies	NOH	Number of hierarchies
Cohesion	CAM	Cohesion among methods in class
Abstraction	ANA	Average number of ancestors
Encapsulation	DAM	Data access metric
Composition	MOA	Measure of aggregation
Inheritance	MFA	Measure of functional abstraction
Messaging	CIS	Class interface size

The improvement in quality can be assessed by comparing the quality before and after refactoring applied to fix a number of code-smells. Hence, the total gain in quality G for each of the considered QMOOD quality attributes q_i before and after refactoring can be easily estimated as:

$$G_{q_i} = q_i' - q_i$$

Where q'_i and q_i represent the value of the quality attribute i after and before refactoring respectively.

We see in Figure 14 that all the quality attributes of both systems are improved after fixing several code-smells from the JDI-Ford system. Understandability is the quality factor that has the highest gain value; whereas the Effectiveness quality factor has the lowest one. This can be explained by the types of fixed code-smells that are known to increase the coupling within classes that heavily affect the quality index calculation of the effectiveness factor. Another reason relates to the types of manually suggested refactoring such as move method, move field, and extract class that are known to have a high impact on coupling, cohesion and the design size in classes that serve to calculate the understandability quality factor. To conclude, fixing the detected code-smells by our tool can lead to better code quality and improve the developers' understandability, the reusability, the flexibility and the effectiveness of the refactored system.

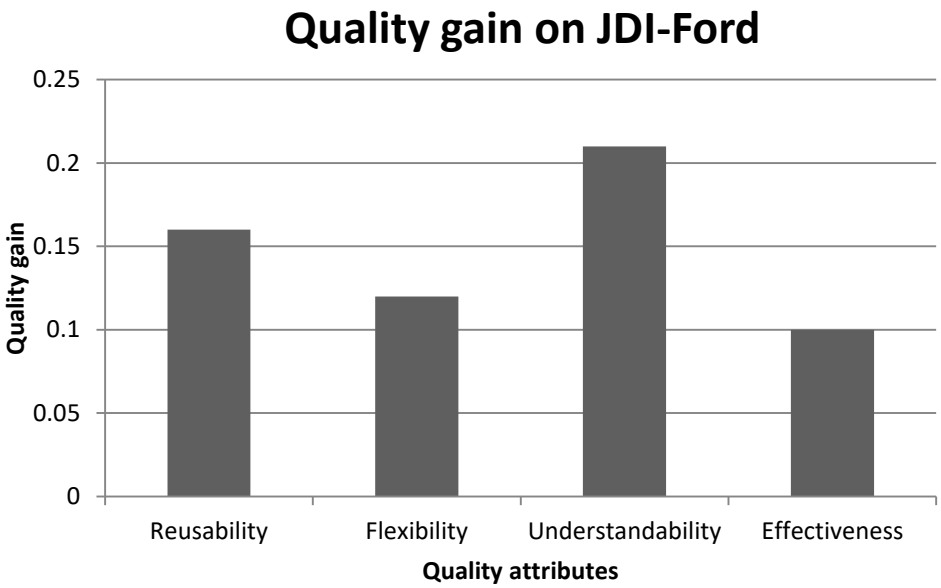


Figure 20 The impact of fixing a number of code-smells (refactorings) on QMOOD quality attributes for JDI-Ford

As described in Table XIII, we asked group D to evaluate the relevance of a random selected set of selected code-smells on the JDI-Ford system. Figure 15 illustrates that only less than 18% of detected code-smells are considered not at all relevant by the software engineers. Around 65% of the code-smells are considered as moderately or extremely relevant by the software developers. This confirms the importance of the detected code-smells for developers that they need to fix them for a better quality of their systems.

It is also important to evaluate the usefulness of the detected code-smells for software maintainers in their daily maintenance activities. To this end, we asked the Ford software developers to justify the usefulness of the code-smells ranked as moderately or extremely relevant. Figure 16 describes the obtained results. The main usefulness is related to refactoring guidance. In fact, most of the software engineers we interviewed found that the detected code-smells give relevant advices about where refactorings should be applied and

what are the common used refactorings to fix these defects. In addition, they found that the code-smells detection process is much more helpful than the traditional analysis of software metrics to find refactoring opportunities. They consider the use of traditional software metrics for Quality Assurance as a time consuming process, and it is easier to interpret the results of detected code-smells and apply the appropriate refactorings to improve the overall quality of the system.

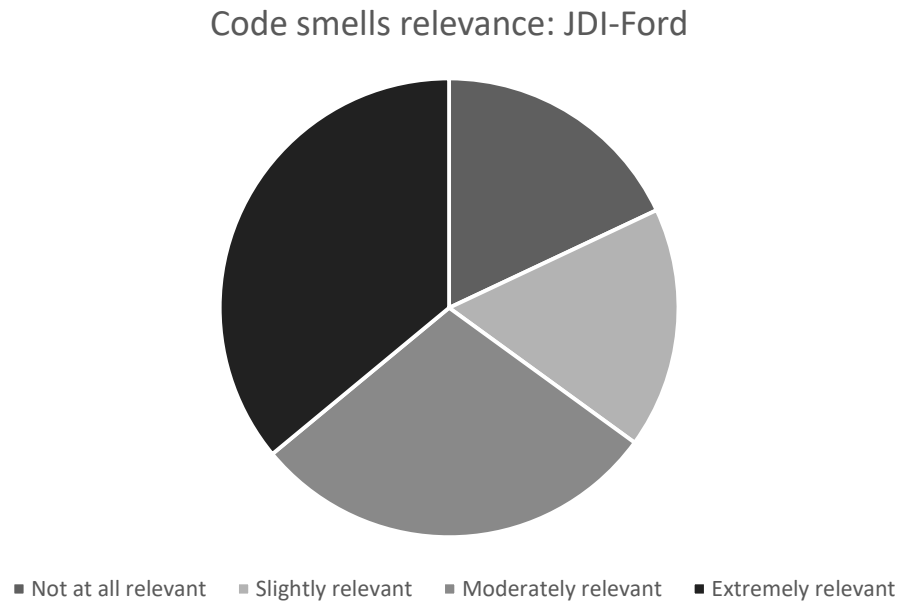


Figure 21 The relevance of detected code-smells on the JDI-Ford system evaluated by the original developers.

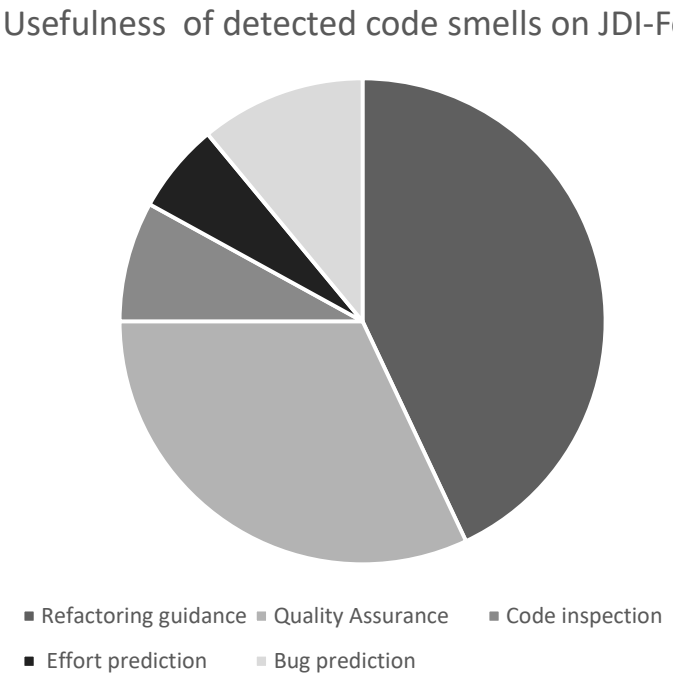


Figure 22 The usefulness of detected code-smells on the JDI-Ford system evaluated by the original developers.

We summarize briefly in the following the feed-back of the Ford developers during the think aloud sessions. Most of the participants mention that the detection rules generated by BLOP represents a faster solution than manual refactoring opportunities detection. The manual techniques represent a time consuming process to find the locations where the refactoring should be applied or to calibrate the metrics threshold or the combination of metrics to identify a maintainability issue manually. The participants found the detection rules useful to maintain a good quality of the design and to make sure that some quality issues are fixed after refactoring. In addition, the developers liked the flexibility to modify the rules (metrics or thresholds) if required. Some possible improvements for our detection techniques were also suggested by the participants. Some participants believe that it will be very helpful to extend the tool by adding a new feature to rank the detected code-smells

based on several criteria such as risk, cost and benefits. They believe that current refactoring tools do not provide any support to estimate the risk, cost and benefits of fixing some maintainability issues. In fact, most of Ford's developers mentioned that they do not fix some code-smells if they feel that it will require a high cost or the benefit is not clear or if the code is stable.

3.4.3.2 Relevance of the Detected Code Smells on the Open Source Systems

We evaluated the relevance of the detected code-smells on the different open source systems by participants of groups A, B and C as described in Table XIII. Figure 17 illustrates that only less than 15% of detected code-smells are considered not at all relevant by the software engineers. Around 70% of the code-smells are considered as moderately or extremely relevant by the different groups, and this confirms the importance of the detected code-smells for developers, and also the results are similar to the industrial case study.

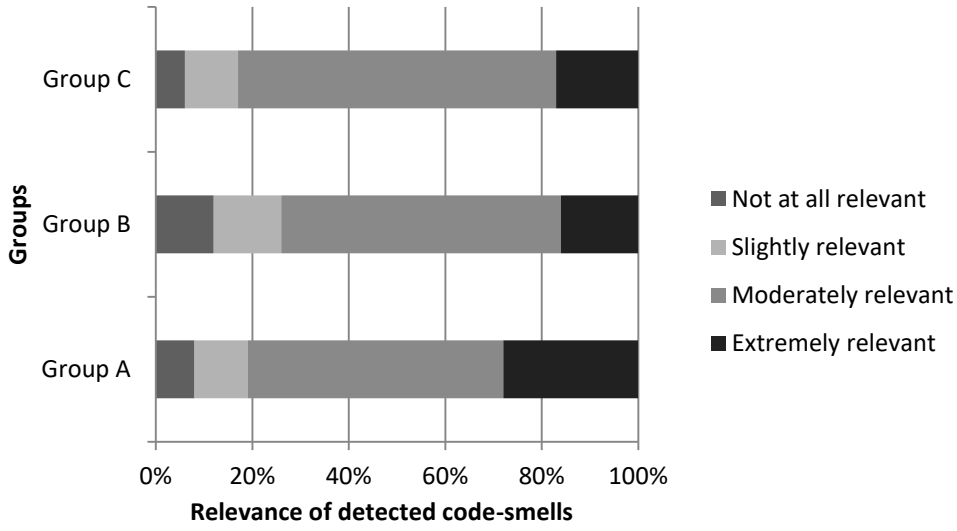


Figure 23 The relevance of detected code-smells on the different open source systems evaluated by developers from groups A, B and C.

To better evaluate the relevance of the detected code-smells, we investigated the types of code-smells that developers perhaps consider them more or less important than others (e.g. Blob, lazy class, etc.). Figure 18 summarizes our findings. It is clear that the detected blobs are considered very relevant for developers. One of the reasons can be the importance of distributing the behavior/functionalities of a program between different classes. In addition, it is very difficult for developers to understand existing functionalities to add new ones if most of them are implemented in one or few classes. Another interesting observation is that lazy classes are not considered very relevant by developers. In fact, lazy classes do not implement in general any important feature thus software engineers did not consider it as an important concern.

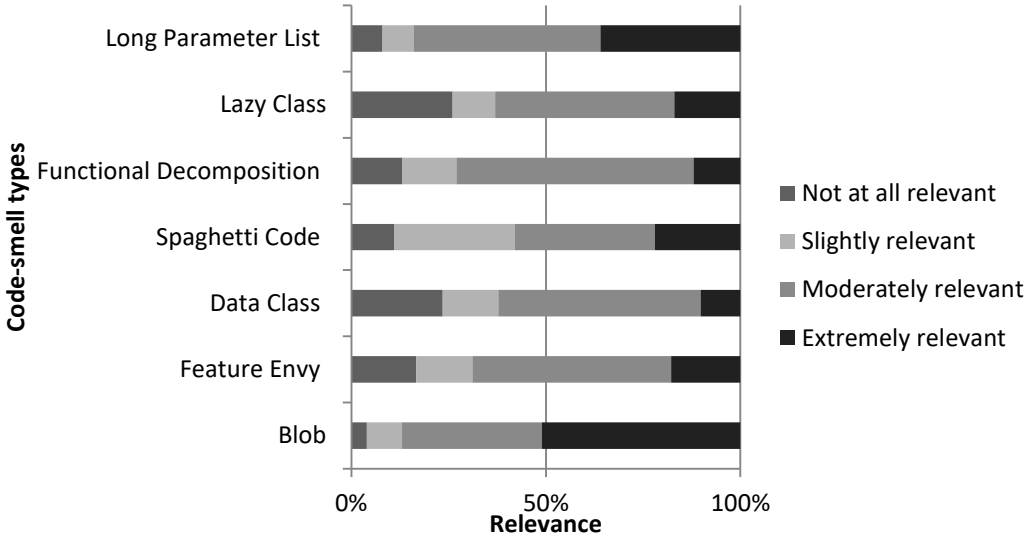


Figure 24 The relevance of the types of detected code-smells on the different open source systems evaluated by developers from the groups A, B and C.

We asked the different groups of software engineers (A-C) to justify the usefulness of the code-smells ranked as moderately or extremely relevant. Figure 19 describes the different reasons of the importance of detected code-smells. Similar to the industrial case study's results, the main usefulness is related to refactoring guidance and quality assurance.

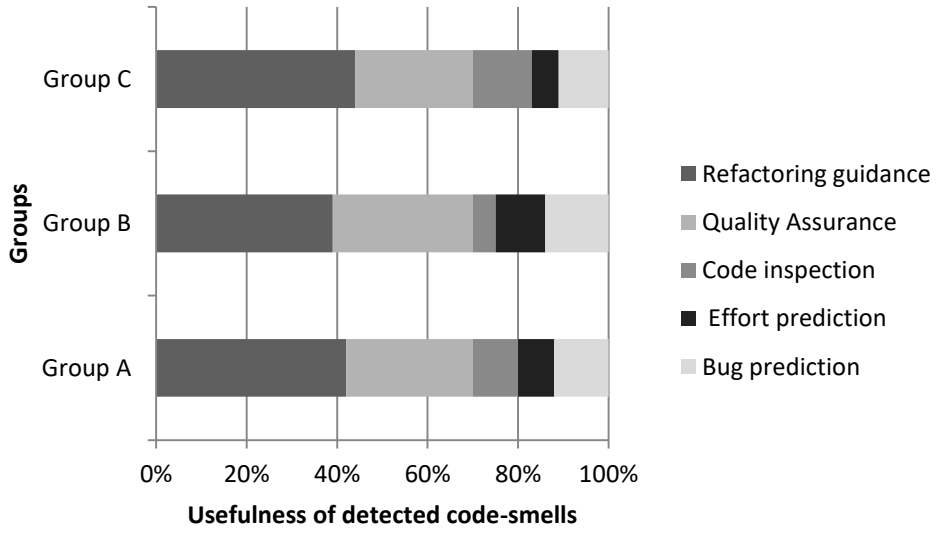


Figure 25 The usefulness of detected code-smells for software maintenance activities on the different open source systems evaluated by developers from the A, B and C groups

3.5 Threats to validity

We explore, in this section, the factors that can bias our empirical study. These factors can be classified in three categories: construct internal and external validity. Construct validity concerns the relation between the theory and the observation. Internal validity concerns possible bias with the results obtained by our proposal. Finally, external validity is related to the generalization of observed results outside the sample instances used in the experiment.

In our experiments, construct validity threats are related to the absence of similar work that uses bi-level techniques for code-smells detection. For that reason, we compare our proposal with two other search-based techniques[7, 98] and DECOR[60]. However, we are aware that there are other tools able to detect several types of code smell that also can be considered in our experiments[24].

Another threat to construct validity arises because, although we considered 7 well-known code-smell types, we did not evaluate the performance of our proposal with other code-smell types. We must further evaluate the performance and ability of our bi-level technique to detect other smells.

A construct threat can also be related to the corpus of manually detected code-smells since developers do not all agree if a candidate is a code-smell or not. We will ask some new experts to extend the existing corpus and provide additional feedback regarding the detected code-smells. In addition, the recall score is challenging to calculate by the software engineers of our experiments and requires additional participants to check its accuracy.

A limitation related to our experiments is the difficulty to set the thresholds for DECOR. In fact, we used the default thresholds used by the DECOR's authors that can have an impact on the quality of the results generated by DECOR.

The evaluation of detected code-smells for some participants is mainly based on the definitions of the code-smells and the examples that we provided during the pilot study. However, the definition of code-smells is subjective and depends on the programming behavior of the participants thus this can affect the accuracy of the detection results.

A construct threat is related to the fact that our detection results depend on the examples of code-smells and well-designed code. In addition, the generation of artificial code fragments can lead to several non-useful examples (generated by the lower-level). Additional constraints should be defined to better guide the search at a lower level to refine the generation of artificial code-smell examples.

The same observation is valid for the used change operators at both the upper and lower levels that can generate invalid rules and code-smell examples (e.g. redundancy) may be avoided by the definition of additional constraints.

We take into consideration the internal threats to validity in the use of stochastic algorithms since our experimental study is performed based on 31 independent simulation runs for each problem instance, and the obtained results are statistically analyzed by using the Wilcoxon rank sum test[100] with a 99% confidence level ($\alpha = 1\%$).

The parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work by additional experiments to evaluate the impact of upper and lower levels' parameters on the quality of the results.

Another internal threat is related to the fact that our tool is not able to rank the detected code-smells. The software engineers considered in our experiments found the type of detected code-smells is helpful to determine its importance. In addition, our contribution in this chapter is mainly related to the detection of code-smells. We will propose several measures that can be used to rank the detected code-smells by our rules. Thus, we consider in our work the detection and ranking of code-smells as two separate steps.

The participants considered in our experiments are not the original developers of the open source systems. Thus, some of their evaluations of the detected code-smells could be not very accurate. In fact, there are, sometimes, good reasons for the design and implementation choices made, and this can be mainly determined by the original developers. However, this is

not the case for the Ford project since some of the original developers of the system participated in our experiments. We are planning to integrate few original developers from these open source projects to evaluate the detected code smells as part of our future work.

Our experiments also lack the evaluation of the impact of removing the detected code-smells on the productivity of the developers and long term maintenance objectives. Another internal threat is the possibility that some developers did not report all the maintainability issues in the evaluated systems thus some detected code-smells are not considered very useful. The use of triangulation as suggested by Yamashita et al.[22] based on the usage of three independent collection methods such as interviews, direct observation and think-aloud sessions may reduce this threat.

The correction of code-smells can be performed by different refactoring strategies thus the quality improvements of the code after fixing some code-smells depends on the applied refactoring. Consequently, further investigation is needed to evaluate the stability of the refactoring results.

For the selection threat, the subject diversity in terms of profile and experience could affect our study. We mitigated the selection threat by giving written guidelines and examples of code-smells already evaluated with arguments and justification. Additionally, each group of subjects evaluated different code-smells from different systems using different techniques/algorithms. Randomization also helps to prevent the learning and fatigue threats. Only few code-smells per system were randomly picked for the evaluation.

Diffusion threat is related to the fact that most of the subjects are from the same university, and the majority know each other. However, they were instructed not to share information about the experience before a certain date.

External validity refers to the generalization of our findings. In this study, we performed our experiments on nine different widely-used open-source systems and one industrial project belonging to different domains and with different sizes. However, we cannot assert that our results can be generalized to other applications, other programming languages, and to other practitioners.

We selected subjects who had similar programming skill levels to reduce the impact of skills on the results, but it is challenging to evaluate the background and skills of the participants objectively. However, a larger number of subjects is required to improve the evaluation of our detection technique. In addition, the manual validation of the detected code-smells is limited to some samples of detected code-smells due to the limited number of participants.

An evaluation of the usefulness of detected code-smells are based only on four maintainability objectives from QMOOD thus the consideration of additional objectives such as performance and the number of bugs after fixing the smells can lead to a better evaluation of the usefulness of detected code-smells. In addition, The quality of detection rules depends on the quality metrics used as input by our bi-level technique that are limited to fourteen metrics in our experiments. Additional metrics may lead to a better quality of the results with other programming languages.

An evaluation of the relevance of detected code-smells in our experiments heavily depends on the opinion of the developers, and it is difficult to generalize due to the limited number of participants in our experiments.

3.6 Conclusion

In this chapter, we have addressed the problem of the absence of consensus in code-smells detection. In fact, choosing quality metrics to detect symptoms of code-smells is not straightforward in software engineering and is usually a challenging task. In order to tackle this problem, we have proposed a bi-level evolutionary optimization approach. The upper-level optimization produces a set of detection rules, which are combinations of quality metrics, with the goal to maximize the coverage of not only a code-smell example base but also a lower-level population of artificial code-smells. The lower-level optimization tries to generate artificial code-smells that cannot be detected by the upper-level detection rules, thereby emphasizing the generation of broad-based and fitter rules. The statistical analysis of the obtained results over nine studied software systems have shown the competitiveness and the outperformance of our proposal in terms of precision and recall over a single-level genetic programming, co-evolutionary, and non-search-based methods.

Following this work, we have identified several avenues for future research. Firstly, the main problem when using bi-level optimization in software engineering is the computational cost required for the lower-level search. Hence, it would be interesting to use regression methods for approximating the lower level optimum for a given upper-level solution. In this

way, we could minimize the required number of function evaluations significantly. Secondly, the idea of bi-level optimization seems interesting for several other SE problems. It would be challenging to model and then solve other interesting SE problems in a bi-level manner. We are currently working on extending our work by proposing a bi-level approach for the correction of code-smells. Finally, our contribution in this chapter is mainly related to the detection of code-smells. We will propose several measures that can be used to rank the detected code-smells by our rules.

Chapter 4: Model Transformation Testing: A Bi-Level Search-Based Software Engineering Approach

4.1 Introduction

Model-Driven Engineering (MDE) [102] considers models as first-class artifacts during the software lifecycle. Available techniques, approaches, and tools for MDE are growing and they support a huge variety of activities, such as model creation, model transformation, and code generation. Especially, model transformations are seen as the heart and soul of MDE[103]. In general, model transformations generate target models from source models which also allow transforming between different modeling languages. Modeling languages are defined with so-called metamodels and on this level the model transformations are described. Thus, model transformations are generalized descriptions able to transform any valid source models conforming to source metamodels into target models conforming to target metamodels. Given the predominant role of model transformations in MDE, efficient techniques and tools for validating model transformations are needed. One of them is model transformation testing [32, 37, 40].

Two important issues must be addressed in model transformation testing: the efficient generation/selection of test cases and the definition of an oracle function to assess the validity of transformed models, and consequently, of the defined rules. This work is concerned with the efficient generation of test cases.

In this thesis, we start from the observation that the generation of a good set of test cases heavily depends not only on the coverage of the metamodels but also on their ability to detect potential errors in the rules. The exploration of the large number of possible transformation possibilities is challenging. Mutation testing is one efficient technique to evaluate the ability of the generated test cases to detect possible errors that sometimes cannot be detected by test cases that ensure a high metamodel coverage due to the high number of possible rules execution. The mutation analysis consists of creating a set of faulty versions, called mutants, of a program with the ultimate goal of designing a test set that distinguishes the program from all its mutants. Both mechanisms for the generation of test cases and the generation of mutants are dependent. Thus, the intuition behind this work is to introduce errors (mutants) in the transformation rules (program) that cannot be detected by some possible test cases solutions, and subsequently, to adapt these test cases to be able to detect the generated mutants. These two steps are repeated until reaching a termination criterion (e.g., number of iterations). To this end, we propose, for the first time, to consider model transformation testing as a bi-level optimization problem[44, 105].

We implemented our proposed bi-level approach and evaluated it on two different ATL transformations [107]. The primary goals of this contribution can be summarized as follows:

(3) We introduce a novel formulation of model transformation testing as a bi-level optimization problem.

(4) We report the results of an empirical study with an implementation of our bi-level approach. The obtained results provide evidence to support the claim that our proposal

is more efficient, on average, than an existing technique based on metamodels coverage[37].

4.2 Approach

This section shows how the above-mentioned issues can be addressed using bi-level optimization and describes the principles that underlie the proposed method for the efficient generation of test cases for ATL transformations.

As described in Figure 5, the upper level process has a main objective which is the generation of test cases that can cover as much as possible the source and target metamodels and detect the errors/mutants introduced by the lower level in the ATL rules. The lower level has one objective which is maximizing the number of generated mutants/errors that cannot be detected by the test cases generated by the upper level. There is a hierarchy in the problem, which arises from the manner in which the two entities operate. In this framework, we observe that the test cases generation process acts as a leader (the important output of the problem), and the mutants generation process acts as a follower.

The overall problem appears as a bi-level optimization task, where for each set of generated test cases (upper level solution), the upper level observes how the lower-level acts by generating errors that cannot be detected by the upper level (leader) test cases and then chooses the best set of test cases which suit it the most, taking the actions of the mutants generation process (lower level or follower) into account. It should be noted that in spite of different objectives appearing in the problem, it is not possible to handle such a problem as a

simple multi-objective optimization task. The reason for this is that the leader cannot evaluate any of its own strategies without knowing the strategy of the follower, which it obtains only by solving a nested optimization problem. The two populations in co-evolution are considered with same importance; however the upper level is more important than the lower level in any bi-level formulation. We will compare later in the experimentation section, with more details, the difference between our bi-level formulation and co-evolution. Next, we describe our adaptation of bi-level optimization to the generation of test cases for model transformations in more details.

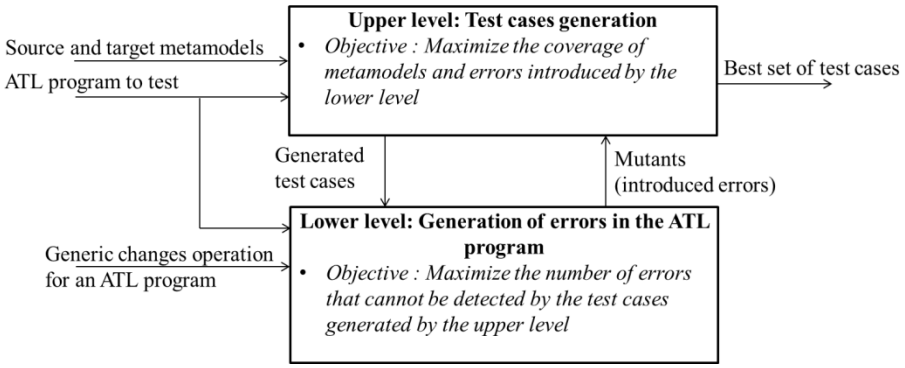


Figure 26 Approach overview

4.2.1 Problem Formulation

The upper level problem involves searching for the best set of test cases among the possible candidate ones, which constitute a huge search space. Thus, a solution is a set of models (test cases) where the goal is cover the source and target metamodels and the errors generated by

the lower level. We used a genetic algorithm (GA)[11] for each level. We describe, in the following, the adaptation of GA for each level.

Our proposed bi-level formulation of the model transformation testing problem is described in Figure 5. For the *upper-level*, we represent a solution (set of test cases) as a vector where each dimension is a source model element. A model element can be a class, attribute, association, or a generalization. The definition depends on the used metamodels. The values of these elements are considered as part of the fitness function that will be described later. For the *lower-level* optimization problem, we implemented a parser to generate an abstract syntax tree from an ATL program (a set of model transformation rules). We also implemented a set of generic change operations that can be applied to any ATL program (Tree). These operations are the following: *updateMetamodelElement(rule, metaModelElement)* consists of modifying a rule by replacing a metamodel element by another one, *deleteMetaModelElement(rule,metaModelElement)* deletes a metamodel element with the associated logic operators (AND/OR), if any, *deleteRule(rule),* and *addNewRule(rule).* A solution at the lower-level is represented as a vector where each dimension is a change operator to modify the original ATL program.

At the upper-level, an objective function is formulated to maximize the coverage of source and target metamodels and also maximize the coverage of generated errors by the best solution found in lower level. Thus, the objective function at the upper level is defined as follows:

$$\text{Maximize } f_{\text{upper level}} = \frac{coverage(testcases, metamodels) + \frac{numberOfDetectedErrors(testcases, rules)}{numberOfErrorsIntroducedByTheLowerLevel}}{2}$$

The method used to derive metamodel coverage was first introduced in[37]. This method begins by a priori performing partition analysis in which the types of coverage criteria taken into consideration for a given problem are chosen. In particular, the approach reasons about the coverage of association instantiation concerning their concrete multiplicities and on the coverage of attribute instantiation by considering their representative values. Each coverage criterion must be partitioned into logical partitions that, when grouped together, represent all the value types each criterion could take on. These partitions are then assigned representative values to represent each coverage criterion partition. After representative values are defined, a set of coverage items for the target metamodel is created. In our adaptation of the coverage item set creation method introduced in[37], we calculate all possible tuple combinations of representative values from all partitions of all coverage criteria types that are included in the target metamodel. The metamodel coverage value for given test case models and target metamodel is determined by calculating the percentage of metamodel coverage items the test case models satisfy. The second part of the upper level fitness function counts the number of error detected by the generated test cases. An error is detected when dissimilarity is found between the target model generated by the original ATL program and the target model generated by the mutants ATL program. If both target models are the same then the test case cannot detect the error in the rules. Both components of the fitness function are normalized in the range [0..1].

At the lower level, the fitness function maximizes the number of introduced errors in the ATL program that cannot be detected by the upper level as defined by the following equation

$$\text{Maximize } f_{\text{lower level}} = \frac{numberOfUndetectedErrors(testcases, rules)}{numberOfGeneratedErrors}$$

It is clear that the evaluation of solutions (test cases) at the upper level depends on the best solutions generated by the lower level (errors). Thus, the fitness function of solutions at the upper level is calculated after the execution of the optimization algorithm in the lower level for each iteration. At the lower level, for each solution (test cases) of the upper level an optimization algorithm is executed to generate the best set of errors that cannot be detected by the test cases at the upper level.

The fitness function of the upper level may get better or worst depending on the new errors introduced to the ATL rules. These errors can be covered or not by the test cases (TC) on the upper level. In fact, the upper-level will continue to execute the genetic algorithm after the new errors introduced by the lower-level. Of course, the upper-level may update some of the good TC since the crossover operator take into-consideration both good and bad quality of parents. In addition, the GA generates part of the population randomly for every iteration to improve the diversity of the solutions.

To generate new solutions for the next iteration ($i+1$) after evaluating them in iteration i, GA uses two operators: crossover and mutation.

For the upper level, the crossover operator allows creating two offspring o_1 and o_2 from the two selected parents p_1 and p_2, as follows:

(4) A random position k is selected in the vector solution.

(5) The first k elements of p_1 become the first k elements of o_1. Similarly, the first k elements of p_2 become the first k elements of o_2.

(6) The remaining elements of, respectively, p_1 and p_2 are added as second parts of, respectively, o_2 and o_1 (hence having a crossing-over operation between parents).

For the lower-level, since the solutions can have variable size we proposed another type of crossover. In fact, several crossover operators were proposed to deal with variable-length chromosomes such as the SAGA (Speciation Adaptation Genetic Algorithm) crossover, the VIV crossover (VIrtual virus), and the SLVC (Synapsing Variable-Length Crossover)[108, 109]. Most of these operators are based on computing the similarity between the two parents, which requires using or defining a meaningful similarity metric. In order to avoid such issue, we use the cut-and-splice crossover [39] which does not require the similarity information. The basic principle of this operator is illustrated by the Figure 6 and is as follows:

(1) A random position k_1 is selected in the first parent vector solution;

(2) A random position k_2 is selected in the second parent vector solution;

(3) The first k_1 elements of the first parent p_1 become the first elements of the first child o_1. Similarly, the first k_2 elements of the second parent p_2 become the first elements of the second child o_2.

(4) The remaining elements of, respectively, p_1 and p_2 are added as second parts of, respectively, o_2 and o_1 (hence having a crossing-over operation between parents).

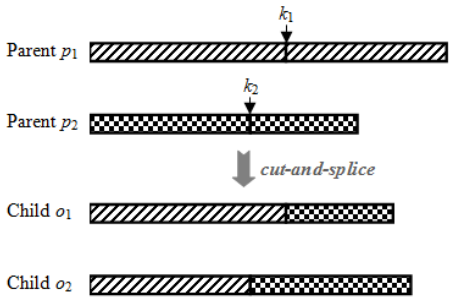


Figure 27 Cut-and-splice crossover principle

The mutation operator consists of randomly changing a dimension in the generated vector. Regarding the validity of the models after applying change operators, we checked, of course, the validity of the generated models with the metamodels using the well-formed rules and metamodel constraints. We apply a repair operator in the case that invalid models are generated (violated the constraints) by selecting another dimension of the vector to modify.

Upper level algorithm: TestCasesGeneration

01. **Inputs:** Source and target metamodels *MM*, ATL program to test (rules) R, Number of best upper solutions that are considered for lower level optimization *nbest*, Upper population size N_1, Lower population size N_2, Upper number of generations G_1, Lower number of generations G_2
02. **Output:** Best set of test cases *T*
03. **Begin**
04. $P_0 \leftarrow$ Initialization (N_1, *MM*);
05. **For each** TC_0 **in** P_0 **do** /**TC means test cases solution*/

06. $ER_0 \leftarrow$ GAErrorsGeneration (TC_0, R, N_2, G_2); */*Call lower level routine*/*

07. $TC_0 \leftarrow$ Evaluation (ER_0, MM, R, TC_0);

08. **End For**

09. $t \leftarrow 1$;

10. **While** ($t < G1$) **do**

11. $Q_t \leftarrow$ Variation ($P_{t\text{-}1}$);

12. **For each** TC_t **in** Q_t **do** */*Evaluate test cases solution based on upper fitness function*/*

13. $TC_t \leftarrow$ UpperEvaluation (TC_t, MM, R);

14. **End For**

15. **For each of the best** *nbests* TC_t **in** Q_t **do** */*Only nbest (best) solutions are used to*/*

16. $ER_t \leftarrow$ GAErrorsGeneration (TC_t, R, N_2, G_2); */*Call lower level routine*/*

17. $TC_t \leftarrow$ EvaluationUpdate (MM, R, TC_t, ER_t); */*Update based on lower level*/*

18. **End For**

19. $U_t \leftarrow P_t \cup Q_t$;

20. $P_{t+1} \leftarrow$ EvironmentalSelection (N_1, N_2, U_t);

21. $t \leftarrow t+1$;

22. **End While**

23. $T \leftarrow$ FittestSelection (P_t);

24. **End**

Figure 28 Pseudo-code of the bi-level adaptation for model transformation testing.

Lower level algorithm: MutantsGeneration

01. **Inputs:** Upper level test cases *TC*, program to test (rules) R, Population size *N*, number of generations *G*
02. **Output:** Best set of mutants MUT
03. **Begin**
04. $P_0 \leftarrow$ Initialization (N, R);
05. $P_0 \leftarrow$ Evaluation (P_0, R, TC); /**Evaluation depends of TC*/*
06. $t \leftarrow 1$;
07. **While** ($t < G$) **do**
08. $Q_t \leftarrow$ Variation (P_{t-1});
09. $Q_t \leftarrow$ Evaluation (Q_t, R, TC); /**Evaluation depends of TC*/*
10. $U_t \leftarrow P_t \cup Q_t$;
11. $P_{t+1} \leftarrow$ EvironmentalSelection (N, N_1, N_2, U_t);
12. $t \leftarrow t+1$;
13. **End While**
14. $MUT \leftarrow$ FittestSelection (P_t);
15. **End**

Figure 29 Pseudo-code of the bi-level adaptation for model transformation testing.

4.2.2 Evaluation

In order to evaluate our approach for model transformations testing using the proposed bi-level optimization (BLOP) approach, we conducted a set of experiments based on two model transformation programs written in ATL[110]. The first ATL program is a set of rules that transforms a class diagram (CLD) to Relational databases (RDBMS) and the second one transforms Sequence Diagrams (SD) to Statechart Diagram (STD). Each experiment is repeated 31 times, and the obtained results are subsequently statistically analyzed with the

aim to compare our bi-level proposal with other transformation testing approaches. In this section, we first present our research questions and then describe and discuss the obtained results. Finally, we present various threats to the validity of our experiments.

4.3 Validation

4.3.1 Research Questions

We defined four research questions that address the applicability, performance comparison with other model transformation testing approaches, and the scalability of our bi-level proposal. The four research questions are as follows:

- *RQ1: How does BLOP perform to generate to generate efficient test cases for model transformations?*
- *RQ2: How does BLOP perform compared to co-evolution (Co-Evol) to generate efficient test cases?*
- *RQ3: How does BLOP perform compared to an existing test cases generation technique not based on the use of metaheuristic search?*
- *RQ5: How does our bi-level formulation scale?*

To answer RQ1, we used two different ATL programs for the transformation of class diagram (CLD) to Relational databases (RDBMS) and the transformation of Sequence Diagrams (SD) to Statechart Diagram (STD). We defined a total of 10 scenarios as described in Table 1. In each scenario, we introduced manually several errors in the ATL program such

modifying, adding and deleting the transformation rules. Then, we evaluate the efficiency of the generated test cases to detect these errors. The four developers (graduate students in Software Engineering at the University of Michigan) considered in our experiments are not aware about the types of errors introduced to the rules by our bi-level approach thus they modified the ATL programs manually to introduce different errors: redundant rules, missing rules (deleting rules), adding faulty rules and modifying existing rules. The developers do not have any idea about the generated errors by the bi-level approach. The developers modified directly the rules without the need to use specific change operators but the bi-level approach used the operators to inject the errors. The change operators modify the original ATL program. Thus, the four developers were responsible for manually changing the ATL code to introduce the errors

The introduced errors by the mutation operators or manually by the developers are formalized as a sequence of operations applied to the ATL program using a Higher-Order Transformation (HOT)[111], i.e., which is a transformation reading a transformation as input and producing a transformation as output. In particular, we reused the fault model and corresponding mutation operators presented in reference[112] and concretized them for ATL. To this end, we define the Precision measure that corresponds to the number of detected errors over the total number of manually introduced ones. An error is detected when dissimilarity is found between the target model generated by the original ATL program and the target model generated by the modified ATL program. If both target models are the same

then the test case cannot detect the error in the rules. The errors reported in Table 1 are those introduced manually by the developers (expected errors to be detected by the test cases).

The reason to use human generated errors is related to the fact that developers modify directly the ATL program without the use of change operators and without knowing any details about the generated errors by the bi-level approach. Thus, we can have an accurate evaluation of the performance of our bi-level approach to generate good test cases. If we generated the errors automatically then maybe they will be easily detected by our test cases. In addition, the developers have an idea about common errors that can happen when writing ATL rules.

To answer RQ2, we compared our BLOP approach to another search-based algorithm called co-evolutionary (Co-Evol) algorithms (due to the absence of other works that use search-based techniques for model transformation mutation analysis). In Co-Evol algorithms the two populations were evolved in parallel without hierarchy. Thus, the second population solutions are independent from the solutions in the first population which is one of the main differences with our bi-level extension.

To answer RQ3, we compared our results with an existing technique for test cases generation not based on heuristic search[37]. In fact, Fleurey et al. proposed a framework for the generation of model transformation test cases based on meta-models coverage. Fleurey et al. algorithm can be just applied to test class diagram to relational schema. They did not adapt for the case of sequence diagram to Statechart Diagram. To answer the last question RQ4, we

evaluated the execution time required by our BLOP proposal based on different scenarios (parameters setting).

Table 13 Scenarios

Scenarios	Modified ATL program	Number of rules of the program	Number of errors
Scenario 1	CLD-to-RDBMS	14	13
Scenario 2	CLD-to-RDBMS	14	11
Scenario 3	CLD-to-RDBMS	14	8
Scenario 4	CLD-to-RDBMS	14	13
Scenario 5	CLD-to-RDBMS	14	9
Scenario 6	SD-to-STD	17	6
Scenario 7	SD-to-STD	17	11
Scenario 8	SD-to-STD	17	14
Scenario 9	SD-to-STD	17	8
Scenario 10	SD-to-STD	17	9

4.3.2 Experimental Settings

For each search algorithm and for each system, we performed a set of experiments using several population sizes: 10, 20, 30, 40 and 50. The stopping criterion was set to 750,000 fitness evaluations for all algorithms in order to ensure fairness of comparison. We used a

high number of evaluations as stopping criterion since our bi-level approach involves two levels of optimization. Each algorithm was executed 31 times with each configuration and then comparison between the configurations was performed based on the precision measure using the Wilcoxon test. Since the replication of the bi-level experiment for 31 times is computationally expensive, the parameter setting experiments are performed on a cluster of 30 machines. For our bi-level approach, both lower-level and upper-level GAs are run each with a population of 30 individuals and 50 generations. The models are generated randomly in the first generation and this explains the high number of iterations used as stopping criterion.

The other parameters' values were fixed by trial and error and are as follows: (1) crossover probability = 0.6; mutation probability = 0.4 where the probability of gene modification is 0.4.

It should be noted that the lower-level routine is not called for all upper-level population members. To control, the high computational cost of our bi-level approach, only *nbest*% of the best upper-level population members are allowed to call the lower-level optimization algorithm. Based on a parametric study, a value of 12% for *nbest* is found to be adequate empirically in our experiments. The *nbest* parametric study will be discussed later. For our experiment, we generated up to 150 test cases and up to 30 errors per solution.

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental

study is performed based on 31 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon signed-rank test[100] with a 95% confidence level ($\alpha = 5\%$). The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples to verify whether their population mean-ranks differ or not. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H_0 while it is true (type I error). A p-value that is less than or equal to α (≤ 0.05) means that we accept H_1 and we reject H_0. However, a p-value that is strictly greater than α (> 0.05) means the opposite. In this way, we could decide whether the outperformance of BLOP over one of each of the others detection algorithms (or the opposite) is statistically significant or just a random result.

The Wilcoxon signed-rank test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference magnitude. The effect size could be computed by using the Cohen's *d* statistic [40]. The effect size is considered: (1) small if $0.2 \leq d < 0.5$; (2) medium if $0.5 \leq d <$ 0:8, or (3) large if $d > 0.8$.

4.3.3 Results and discussions

Results for RQ1. In this section, we evaluate the performance of our BLOP adaptation on the generation of efficient test cases for ATL programs. Figures 8 and 9 summarize our findings. The expected errors were detected with an average of more than 90% of precision on the 10 different scenarios. For over half the total number of scenarios, 100% precision was obtained, indicating the detection of all expected errors. We noticed that our technique

does not have a bias towards the types of errors that are introduced. As described in Figures 8 and 9, in all systems, we had detected most of the introduced errors. To conclude, our BLOP approach is able to generate efficient test cases able to detect errors in ATL programs (RQ1).

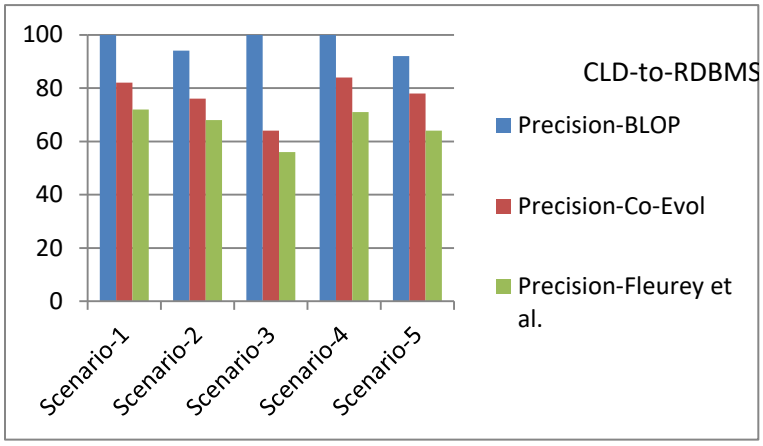


Figure 30 Precision median values of BLOP, Co-Evol and Fleurey et al. [28] over 31 independent simulation runs.

Results for RQ2. In this section, we compare our BLOP adaptation to another search- based techniques based on co-evolution algorithms (Co-Evol). Figures 8 and 9 show the overview of the results of the significance tests comparison between these algorithms. It is clear that BLOP outperforms Co-Evol in 100% of the scenarios in terms of precision. For Co-Evol, the two populations are executed in parallel and the problem is that there is no dependency between both populations (unlike BLOP that creates a hierarchy between two levels) thus one population can converge before the second one. We found that the main reason explaining the outperformance of BLOP against Co-Evol is the diversity of the generated errors. In fact, the lower level of our BLOP formulation generates mutants/errors for every

good solution (test cases) in the upper level then these errors are used to evaluate the solutions in the upper level. Thus, the generated errors depend on the associated solution (test cases) in the upper level.

There is no parallelism in our bi-level formulation in contradiction to co-evolutionary algorithms. The upper level is executed for number iterations then the lower level for another number of iterations. In co-evolutionary algorithms, populations are executed in parallel without hierarchy (but can be dependent such as by exchanging information[113]). The problem with the Co-Evolutionary approach is that one population may converge before the other. Contrariwise, in our bi-level approach there is a hierarchy that allows avoiding the problem of premature convergence of one population over the other. Indeed, the evaluation of every good test cases solution (upper level) requires the running a search algorithm to find the best undetectable ATL errors by the upper level solution. This concept avoids driving the search towards uninteresting directions. Furthermore, the two populations in co-evolution are considered with same importance; however the upper level is more important than the lower level in any bi-level formulation.

In conclusion, we answer RQ2 by concluding that the results in our experiments confirm that our proposed BLOP is adequate and it outperforms Co-Evol.

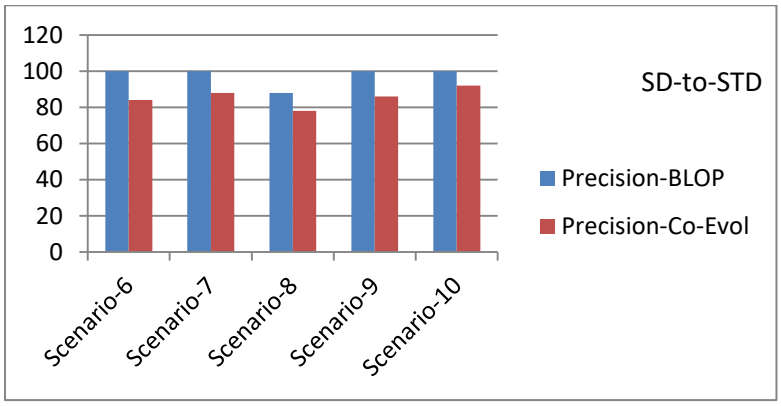


Figure 31 Precision median values of BLOP and Co-Evol over 31 independent simulation runs.

Finally, the following table gives the effect sizes in addition to the p-values of the Wilcoxon test.

Table 14 The Wilcoxon test p-values and the effect size values (Cohen's d statistic) of the comparisons between BLOP and Co-Evol on the 10 scenarios.

Scenario	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
p-value	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Effect size	0.83	0.79	0.82	0.86	0.72	0.68	0.66	0.73	0.36	0.38

Results for RQ3. Since it is not sufficient to compare our proposal with only search-based work, we compared the performance of BLOP with the test cases generation techniques proposed by Fleurey et al. [37] based only on metamodel coverage. Figure 8 summarizes the results of the precision obtained on the CLD-to-RDBMS ATL program. It is clear from the results of Figure 8 that the metamodel coverage criterion is not sufficient and our BLOP technique generates more efficient test cases than Fleurey et al. [37]. To conclude, our BLOP

adaption also outperforms, on average, an existing approach not based on meta-heuristic search (RQ3).

Results for RQ4. Since our proposal is based on bi-level optimization, it is important to evaluate the execution time. It is evident that BLOP requires higher execution time than Co-Evol and[37] since BLOP has an optimization algorithm to be executed at the lower level. To reduce the computational complexity of our BLOP adaptation, we selected only best solutions (nbest%) at the upper level to update their fitness evaluations based on the coverage of errors that are generated by the optimization algorithms executed at the lower level for every selected solution. All the algorithms under comparison were executed on machines with Intel Xeon 3 GHz processors and 4 GB RAM. This allows us to make a fair comparison between CPU times. The average execution time for BLOP, was around 3.5 hours for a single execution of the algorithm with a total of 750K iterations. An important parameter that reduced the execution time of our BLOP adaptation is the number of selected good solutions at the upper level. Figure 8 shows that the performance of our approach improves as we increase the percentage of best solutions selected from the upper level for each iteration. However, the results become stable after 12% (percentage of selected solutions from the upper level population). For this reason, we considered this threshold in our experiments that represents a good trade-off between the quality of test cases solutions and the execution time. As described in Figure 10, the average precision scores on the 10 scenarios become almost stable after the 12% threshold value and the execution time increases dramatically since a high number of optimization algorithms are executed at the lower level.

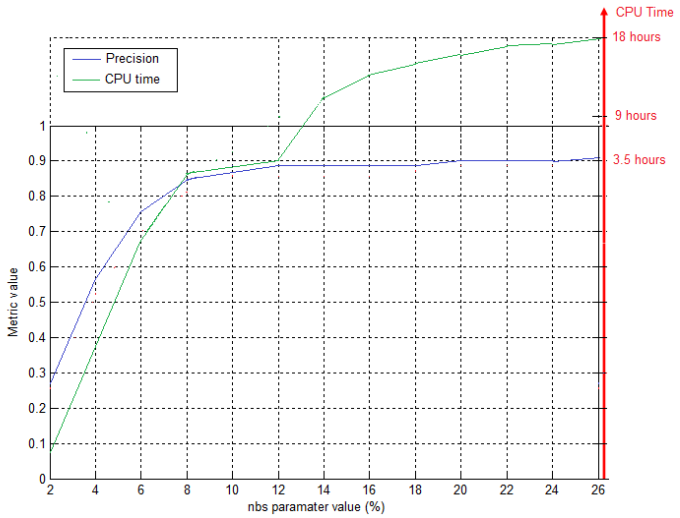


Figure 32 Scalability of our bi-level approach for test cases generation

4.3.4 Threats to Validity

We considered different types of threat that can affect the validity of our experiments. We consider each of these in the following paragraphs.

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. We used the Wilcoxon signed-rank test with a 95% confidence level to test if significant differences existed between the measurements for different treatments. This test makes no assumption that the data is normally distributed and is suitable for ordinal data, so we can be confident that the statistical relationships we observed are significant. In addition, since the tools are not available we re-implemented the existing transformation techniques for a comparison with our proposal thus the outperformance of our bi-level can be also related to some missing implementation details of existing approaches.

Internal validity is concerned with the causal relationship between the treatment and the outcome. When we observe an increase in precision, was it caused by our bi-level approach, or could it have occurred for another reason? We dealt with internal threats to validity by performing 31 independent simulation runs for each problem instance. This makes it highly unlikely that the observed increased in precision was caused by anything other than the applied bi-level approach.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on two ATL programs belonging to different domains and with different sizes. However, we cannot assert that our results can be generalized to other transformation mechanisms. Future replications of this study are necessary to confirm our findings. In addition, our experiments are limited to only 10 scenarios. We plan to extend the experiments by considering additional possible errors in the ATL programs.

Chapter 5: Conclusion and Future Work

Choosing quality metrics to detect symptoms of code-smells is not straightforward in software engineering and is usually a challenging task. In order to tackle this problem, we have proposed a bi-level evolutionary optimization approach as our first contribution. The upper-level optimization produces a set of detection rules, which are combinations of quality metrics, with the goal to maximize the coverage of not only a code-smell example base but also a lower-level population of artificial code-smells. The lower-level optimization tries to generate artificial code-smells that cannot be detected by the upper-level detection rules, thereby emphasizing the generation of broad-based and fitter rules. The statistical analysis of the obtained results over nine studied software systems have shown the competitiveness and the outperformance of our proposal in terms of precision and recall over a single-level genetic programming, co-evolutionary, and non-search-based methods. . A bi-level approach for the correction of code-smells is going to be our future work for our first contribution. Finally, our first contribution in this thesis is mainly related to the detection of code-smells.

As our second contribution, we proposed a bi-level evolutionary optimization approach for model transformation testing. The upper-level optimization produces a set of test cases, which are models, conformed to a meta-model, with the goal to maximize the coverage of metamodels and the lower-level population of errors. The lower-level optimization tries to generate errors that cannot be detected by the upper-level test cases, thereby emphasizing the generation of broad-based and fitter test cases. The statistical analysis of the obtained results

has shown the competitiveness and the outperformance of our proposal in terms of precision over co-evolutionary and non-search-based methods.

Following these two contribution, we have identified several avenues for future research. Firstly, the main problem when using bi-level optimization in software engineering is the computational cost required for the lower-level search. Hence, it would be interesting to use regression methods for approximating the lower level optimum for a given upper-level solution. In this way, we could minimize the required number of function evaluations significantly. Secondly, the idea of bi-level optimization seems interesting for several other SE problems. It would be challenging to model and then solve other interesting SE problems in a bi-level manner.

Bibliography

1. Liu, H.; Guo, X.; Shao, W., Monitor-based instant software refactoring. *IEEE Transactions on Software Engineering* **2013,** *39* (8), 1112-1126.

2. Mens, T.; Tourwé, T., A survey of software refactoring. *IEEE Transactions on software engineering* **2004,** *30* (2), 126-139.

3. Opdyke, W. F. Refactoring object-oriented frameworks. University of Illinois at Urbana-Champaign, 1992.

4. Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D., Refactoring: Improving the design of existing programs. Addison-Wesley Reading: 1999.

5. Fenton, N.; Bieman, J., *Software metrics: a rigorous and practical approach*. CRC Press: 2014.

6. Brown, W. H.; Malveau, R. C.; McCormick, H. W.; Mowbray, T. J., *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.: 1998.

7. Kessentini, M.; Kessentini, W.; Sahraoui, H.; Boukadoum, M.; Ouni, A. In *Design Defects Detection and Correction by Example*, Program Comprehension (ICPC), 2011 IEEE 19th International Conference on, 22-24 June 2011; 2011; pp 81-90.

8. Munro, M. J. In *Product metrics for automatic identification of" bad smell" design problems in java source-code*, 11th IEEE International Software Metrics Symposium (METRICS'05), IEEE: 2005; pp 15-15.

9. Mantyla, M. V. In *Empirical software evolvability-code smells and human evaluations*, Software Maintenance (ICSM), 2010 IEEE International Conference on, IEEE: 2010; pp 1-6.

10. Harman, M.; Mansouri, S. A.; Zhang, Y., Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.* **2012,** *45* (1), 1-61.

11. Goldberg, D. E.; Deb, K., A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms* **1991,** *1*, 69-93.

12. e Abreu, F. B.; Melo, W. In *Evaluating the impact of object-oriented design on software quality*, Software Metrics Symposium, 1996., Proceedings of the 3rd International, IEEE: 1996; pp 90-99.

13. Riel, A. J., *Object-oriented design heuristics*. Addison-Wesley Longman Publishing Co., Inc.: 1996.

14. Coad, P.; Yourdon, E., *Object-oriented design*. Yourdon press Englewood Cliffs, NJ: 1991; Vol. 92.

15. Zhang, M.; Hall, T.; Baddoo, N., Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: research and practice* **2011,** *23* (3), 179-202.

16. Van Emden, E.; Moonen, L. In *Java quality assurance by detecting code smells*, Reverse Engineering, 2002. Proceedings. Ninth Working Conference on, IEEE: 2002; pp 97-106.

17. Mäntylä, M. V.; Lassenius, C., Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering* **2006,** *11* (3), 395-431.

18. Monden, A.; Nakae, D.; Kamiya, T.; Sato, S.-i.; Matsumoto, K.-i. In *Software quality analysis by code clones in industrial legacy software*, Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on, IEEE: 2002; pp 87-94.

19. Deligiannis, I.; Shepperd, M.; Roumeliotis, M.; Stamelos, I., An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software* **2003,** *65* (2), 127-139.

20. Anda, B. In *Assessing software system maintainability using structural measures and expert assessments*, 2007 IEEE International Conference on Software Maintenance, IEEE: 2007; pp 204-213.

21. Rapu, D.; Ducasse, S.; Gîrba, T.; Marinescu, R. In *Using history information to improve design flaws detection*, Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on, IEEE: 2004; pp 223-232.

22. Yamashita, A., Assessing the capability of code smells to support software maintainability assessments: Empirical inquiry and methodological approach. **2012**.

23. Brown, W. H.; Malveau, R. C.; Mowbray, T. J., AntiPatterns: refactoring software, architectures, and projects in crisis. **1998**.

24. Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; De Lucia, A.; Poshyvanyk, D. In *Detecting bad smells in source code using change history information*, Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, IEEE: 2013; pp 268-278.

25. Chidamber, S. R.; Kemerer, C. F., A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on* **1994,** *20* (6), 476-493.

26. e Abreu, F. B. In *The MOOD metrics set*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Workshop on Metrics, Vol. 95, p. 267, 1995; p 267.

27. Abran, A.; Nguyenkim, H., Measurement of the maintenance process from a demand-based perspective. *Journal of Software Maintenance: Research and Practice* **1993,** *5* (2), 63-90.

28. Briand, L. C.; Daly, J. W.; Wust, J. K., A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on software Engineering* **1999,** *25* (1), 91-121.

29. Mäntylä, M.; Vanhanen, J.; Lassenius, C. In *A taxonomy and an initial empirical study of bad smells in code*, Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on, IEEE: 2003; pp 381-384.

30. Khomh, F.; Vaucher, S.; Guéhéneuc, Y.-G.; Sahraoui, H. In *A bayesian approach for the detection of code and design smells*, 2009 Ninth International Conference on Quality Software, IEEE: 2009; pp 305-314.

31. Mantyla, M.; Vanhanen, J.; Lassenius, C. In *A taxonomy and an initial empirical study of bad smells in code*, Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on, IEEE: 2003; pp 381-384.

32. Baudry, B.; Ghosh, S.; Fleurey, F.; France, R.; Le Traon, Y.; Mottu, J.-M., Barriers to systematic model transformation testing. *Communications of the ACM* **2010,** *53* (6), 139-143.

33. Selim, G. M.; Cordy, J. R.; Dingel, J. In *Model transformation testing: The state of the art*, Proceedings of the First Workshop on the Analysis of Model Transformations, ACM: 2012; pp 21-26.

34. Vallecillo, A.; Gogolla, M.; Burgueno, L.; Wimmer, M.; Hamann, L., Formal specification and testing of model transformations. In *Formal Methods for Model-Driven Engineering*, Springer: 2012; pp 399-437.

35. Burgueno, L.; Troya, J.; Wimmer, M.; Vallecillo, A., Static fault localization in model transformations. *IEEE Transactions on Software Engineering* **2015,** *41* (5), 490-506.

36. Ehrig, K.; Küster, J. M.; Taentzer, G., Generating instance models from meta models. *Software & Systems Modeling* **2009,** *8* (4), 479-500.

37. Fleurey, F.; Baudry, B.; Muller, P.-A.; Le Traon, Y., Qualifying input test data for model transformations. *Software & Systems Modeling* **2009,** *8* (2), 185-203.

38. González, C. A.; Cabot, J. In *ATLTest: a white-box test generation approach for ATL transformations*, International Conference on Model Driven Engineering Languages and Systems, Springer: 2012; pp 449-464.

39. Guerra, E. In *Specification-driven test generation for model transformations*, International Conference on Theory and Practice of Model Transformations, Springer: 2012; pp 40-55.

40. Mottu, J.-M.; Baudry, B.; Le Traon, Y. In *Mutation analysis testing for model transformations*, European Conference on Model Driven Architecture-Foundations and Applications, Springer: 2006; pp 376-390.

41. Fraternali, P.; Tisi, M. In *Mutation analysis for model transformations in atl*, Model Transformation with ATL Workshop (MtATL2009), 2009; pp 145-149.

42. Giner, P.; Pelechano, V. In *Test-driven development of model transformations*, International Conference on Model Driven Engineering Languages and Systems, Springer: 2009; pp 748-752.

43. Kolstad, C. D. *A review of the literature on bi-level mathematical programming*; 1985.

44. Sinha, A.; Malo, P.; Deb, K., Efficient evolutionary algorithm for single-objective bilevel optimization. *arXiv preprint arXiv:1303.3901* **2013**.

45. Bracken, J.; McGill, J. T., Mathematical programs with optimization problems in the constraints. *Operations Research* **1973,** *21* (1), 37-44.

46. Bard, J. F.; Falk, J. E., An explicit solution to the multi-level programming problem. *Computers & Operations Research* **1982,** *9* (1), 77-100.

47. Mathieu, R.; Pittard, L.; Anandalingam, G., Genetic algorithm based approach to bi-level linear programming. *Revue française d'automatique, d'informatique et de recherche opérationnelle. Recherche opérationnelle* **1994,** *28* (1), 1-21.

48. Vicente, L. N.; Calamai, P. H., Bilevel and multilevel programming: A bibliography review. *Journal of Global optimization* **1994,** *5* (3), 291-306.

49. Candler, W.; Townsley, R., A linear two-level programming problem. *Computers & Operations Research* **1982,** *9* (1), 59-76.

50. Bialas, W.; Karwan, M.; Shaw, J., A parametric complementary pivot approach for two-level linear programming. *State University of New York at Buffalo* **1980,** *57*.

51. Aiyoshi, E.; Shimizu, K., Hierarchical decentralized systems and its new solution by a barrier method. *IEEE Transactions on Systems, Man and Cybernetics* **1981,** (6), 444-449.

52. Colson, B.; Marcotte, P.; Savard, G., A trust-region method for nonlinear bilevel programming: algorithm and computational experience. *Computational Optimization and Applications* **2005,** *30* (3), 211-227.

53. Legillon, F.; Liefooghe, A.; Talbi, E.-G. In *Cobra: A cooperative coevolutionary algorithm for bi-level optimization*, 2012 IEEE Congress on Evolutionary Computation, IEEE: 2012; pp 1-8.

54. Koh, A., A metaheuristic framework for bi-level programming problems with multi-disciplinary applications. In *Metaheuristics for Bi-level Optimization*, Springer: 2013; pp 153-187.

55. Harman, M. In *The current state and future of search based software engineering*, 2007 Future of Software Engineering, IEEE Computer Society: 2007; pp 342-357.

56. Ó Cinnéide, M.; Tratt, L.; Harman, M.; Counsell, S.; Hemati Moghadam, I. In *Experimental assessment of software metrics using automated refactoring*, Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement, ACM: 2012; pp 49-58.

57. Ciupke, O. In *Automatic detection of design problems in object-oriented reengineering*, Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30 Proceedings, IEEE: 1999; pp 18-32.

58. Kothari, S. C.; Bishop, L.; Sauceda, J.; Daugherty, G., A pattern-based framework for software anomaly detection. *Software Quality Journal* **2004,** *12* (2), 99-120.

59. Dhambri, K.; Sahraoui, H.; Poulin, P. In *Visual detection of design anomalies*, Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on, IEEE: 2008; pp 279-283.

60. Moha, N.; Gueheneuc, Y.-G.; Duchien, L.; Le Meur, A.-F., DECOR: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on* **2010,** *36* (1), 20-36.

61. Salehie, M.; Li, S.; Tahvildari, L. In *A metric-based heuristic framework to detect object-oriented design flaws*, 14th IEEE International Conference on Program Comprehension (ICPC'06), IEEE: 2006; pp 159-168.

62. Sjøberg, D. I.; Yamashita, A.; Anda, B. C.; Mockus, A.; Dybå, T., Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering* **2013,** *39* (8), 1144-1156.

63. Harman, M.; Tratt, L., Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, ACM: London, England, 2007; pp 1106-1113.

64. Harman, M.; Mansouri, S. A.; Zhang, Y., Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* **2012,** *45* (1), 11.

65. Genero, M.; Piattini, M.; Calero, C. In *Empirical validation of class diagram metrics*, Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n, IEEE: 2002; pp 195-203.

66. Kafura, D.; Reddy, G. R., The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering* **1987,** *13* (3), 335.

67. Kataoka, Y.; Notkin, D.; Ernst, M. D.; Griswold, W. G. In *Automated support for program refactoring using invariants*, Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), IEEE Computer Society: 2001; p 736.

68. Liu, H.; Yang, L.; Niu, Z.; Ma, Z.; Shao, W. In *Facilitating software refactoring with appropriate resolution order of bad smells*, Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ACM: 2009; pp 265-268.

69. Siy, H.; Votta, L. In *Does the modern code inspection have value?*, Proceedings of the IEEE international Conference on Software Maintenance (ICSM'01), IEEE Computer Society: 2001; p 281.

70. Schulze, S.; Lochau, M.; Brunswig, S. In *Implementing refactorings for FOP: lessons learned and challenges ahead*, Proceedings of the 5th International Workshop on Feature-Oriented Software Development, ACM: 2013; pp 33-40.

71. Bavota, G.; Gethers, M.; Oliveto, R.; Poshyvanyk, D.; Lucia, A. d., Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **2014,** *23* (1), 4.

72. Bavota, G.; De Lucia, A.; Marcus, A.; Oliveto, R. In *Software re-modularization based on structural and semantic metrics*, Reverse Engineering (WCRE), 2010 17th Working Conference on, IEEE: 2010; pp 195-204.

73. Maletic, J. I.; Marcus, A. In *Supporting program comprehension using semantic and structural information*, Proceedings of the 23rd International Conference on Software Engineering, IEEE Computer Society: 2001; pp 103-112.

74. Kuhn, A.; Ducasse, S.; Gírba, T., Semantic clustering: Identifying topics in source code. *Information and Software Technology* **2007,** *49* (3), 230-243.

75. Scanniello, G.; D'Amico, A.; D'Amico, C.; D'Amico, T., Architectural layer recovery for software system understanding and evolution. *Software: Practice and Experience* **2010,** *40* (10), 897-916.

76. Banerjee, J.; Kim, W.; Kim, H.-J.; Korth, H. F., *Semantics and implementation of schema evolution in object-oriented databases*. ACM: 1987; Vol. 16.

77. Ganter, B.; Wille, R., *Formal concept analysis: mathematical foundations*. Springer Science & Business Media: 2012.

78. Snelting, G.; Tip, F., *Reengineering class hierarchies using concept analysis*. ACM: 1998; Vol. 23.

79. Weiser, M. In *Program slicing*, Proceedings of the 5th international conference on Software engineering, IEEE Press: 1981; pp 439-449.

80. Griswold, W. G.; Chen, M. I.; Bowdidge, R. W.; Morgenthaler, J. D., *Tool support for planning the restructuring of data abstractions in large systems*. ACM: 1996; Vol. 21.

81. Roberts, D. B.; Johnson, R., *Practical analysis for refactoring*. University of Illinois at Urbana-Champaign: 1999.

82. Steimann, F.; Thies, A. In *From public to private to absent: Refactoring Java programs under constrained accessibility*, European Conference on Object-Oriented Programming, Springer: 2009; pp 419-443.

83. Schäfer, M.; Ekman, T.; De Moor, O., Sound and extensible renaming for Java. *ACM Sigplan Notices* **2008,** *43* (10), 277-294.

84. Schäfer, M.; Verbaere, M.; Ekman, T.; de Moor, O. In *Stepping stones over the refactoring rubicon*, European Conference on Object-Oriented Programming, Springer: 2009; pp 369-393.

85. Sahraoui, H. A.; Godin, R.; Miceli, T. In *Can metrics help to bridge the gap between the improvement of oo design quality and its automation?*, Software Maintenance, 2000. Proceedings. International Conference on, IEEE: 2000; pp 154-162.

86. Zhang, S.; Saff, D.; Bu, Y.; Ernst, M. D. In *Combined static and dynamic automated test generation*, Proceedings of the 2011 International Symposium on Software Testing and Analysis, ACM: 2011; pp 353-363.

87. Qayum, F.; Heckel, R. In *Local search-based refactoring as graph transformation*, Search Based Software Engineering, 2009 1st International Symposium on, IEEE: 2009; pp 43-46.

88. Mancoridis, S.; Mitchell, B. S.; Rorres, C.; Chen, Y.-F.; Gansner, E. R. In *Using Automatic Clustering to Produce High-Level System Organizations of Source Code*, IWPC, Citeseer: 1998; pp 45-52.

89. Mitchell, B. S.; Mancoridis, S., On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering* **2006,** *32* (3), 193-208.

90. Seng, O.; Stammel, J.; Burkhart, D. In *Search-based determination of refactorings for improving the class structure of object-oriented systems*, Proceedings of the 8th annual conference on Genetic and evolutionary computation, ACM: 2006; pp 1909-1916.

91. O'Keeffe, M.; Ó Cinnéide, M., Search-based refactoring for software maintenance. *Journal of Systems and Software* **2008,** *81* (4), 502-516.

92. Abdeen, H.; Ducasse, S.; Sahraoui, H.; Alloui, I. In *Automatic package coupling and cycle minimization*, Reverse Engineering, 2009. WCRE'09. 16th Working Conference on, IEEE: 2009; pp 103-112.

93. Ouni, A.; Kessentini, M.; Sahraoui, H.; Boukadoum, M., Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering* **2012,** *20* (1), 47-79.

94. Sayyad, A. S.; Menzies, T.; Ammar, H. In *On the value of user preferences in search-based software engineering: a case study in software product lines*, Software engineering (ICSE), 2013 35th international conference on, IEEE: 2013; pp 492-501.

95. Mkaouer, M. W.; Kessentini, M.; Bechikh, S.; Deb, K.; Ó Cinnéide, M. In *High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III*, Proceedings of the 2014 conference on Genetic and evolutionary computation, ACM: 2014; pp 1263-1270.

96. Bennett, K. H.; Rajlich, V. T. In *Software maintenance and evolution: a roadmap*, Proceedings of the Conference on the Future of Software Engineering, ACM: 2000; pp 73-87.

97. Golberg, D. E., Genetic algorithms in search, optimization, and machine learning. *Addion wesley* **1989,** *1989*, 102.

98. Boussaa, M.; Kessentini, W.; Kessentini, M.; Bechikh, S.; Chikha, S. B. In *Competitive coevolutionary code-smells detection*, International Symposium on Search Based Software Engineering, Springer: 2013; pp 50-65.

99. Bansiya, J.; Davis, C. G., A hierarchical model for object-oriented design quality assessment. *Software Engineering, IEEE Transactions on* **2002,** *28* (1), 4-17.

100. Arcuri, A.; Fraser, G., Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* **2013,** *18* (3), 594-623.

101. Holm, S., A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* **1979**, 65-70.

102. Brambilla, M.; Cabot, J.; Wimmer, M., Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering* **2012,** *1* (1), 1-182.

103. Sendall, S.; Kozaczynski, W. *Model transformation the heart and soul of model-driven software development*; 2003.

104. Lin, Y.; Zhang, J.; Gray, J., A testing framework for model transformations. In *Model-driven software development*, Springer: 2005; pp 219-236.

105. Sahin, D.; Kessentini, M.; Bechikh, S.; Deb, K., Code-smell detection as a bilevel problem. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **2014,** *24* (1), 6.

106. Sumalee, A., Optimal road user charging cordon design: a heuristic optimization approach. *Computer‐Aided Civil and Infrastructure Engineering* **2004,** *19* (5), 377-392.

107. Jouault, F.; Kurtev, I. In *Transforming models with ATL*, satellite events at the MoDELS 2005 Conference, Springer: 2005; pp 128-138.

108. Ryerkerk, M.; Averill, R.; Deb, K.; Goodman, E. In *Meaningful representation and recombination of variable length genomes*, Proceedings of the 14th annual conference companion on Genetic and evolutionary computation, ACM: 2012; pp 1471-1472.

109. Hutt, B.; Warwick, K., Synapsing variable-length crossover: Meaningful crossover for variable-length genomes. *IEEE transactions on evolutionary computation* **2007,** *11* (1), 118-131.

110. Website:, A. P., *https://www.eclipse.org/atl*.

111. Tisi, M.; Jouault, F.; Fraternali, P.; Ceri, S.; Bézivin, J. In *On the use of higher-order model transformations*, European Conference on Model Driven Architecture-Foundations and Applications, Springer: 2009; pp 18-33.

112. Sen, S.; Baudry, B.; Mottu, J.-M. In *On combining multi-formalism knowledge to select models for model transformation testing*, 2008 1st International Conference on Software Testing, Verification, and Validation, IEEE: 2008; pp 328-337.

113. Ren, J.; Harman, M.; Di Penta, M. In *Cooperative co-evolutionary optimization of software project staff assignments and job scheduling*, International Symposium on Search Based Software Engineering, Springer: 2011; pp 127-141.