A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection

Wael Kessentini, *Student Member, IEEE*, Marouane Kessentini, *Member, IEEE*, Houari Sahraoui, *Member, IEEE*, Slim Bechikh, *Member, IEEE*, and Ali Ouni, *Student Member, IEEE*

Abstract—We propose in this paper to consider code-smells detection as a distributed optimization problem. The idea is that different methods are combined in parallel during the optimization process to find a consensus regarding the detection of code-smells. To this end, we used Parallel Evolutionary algorithms (P-EA) where many evolutionary algorithms with different adaptations (fitness functions, solution representations, and change operators) are executed, in a parallel cooperative manner, to solve a common goal which is the detection of code-smells. An empirical evaluation to compare the implementation of our cooperative P-EA approach with random search, two single population-based approaches and two code-smells detection techniques that are not based on meta-heuristics search. The statistical analysis of the obtained results provides evidence to support the claim that cooperative P-EA is more efficient and effective than state of the art detection approaches based on a benchmark of nine large open source systems where more than 85 percent of precision and recall scores are obtained on a variety of eight different types of code-smells.

Index Terms—Search-based software engineering, code-smells, software quality, distributed evolutionary algorithms

1 INTRODUCTION

SOURCE code of large systems is iteratively refined, restructured and evolved due to many reasons such as correcting errors in design, modifying a design to accommodate changes in requirements, and modifying a design to enhance existing features. Many studies reported that these software maintenance activities consume up to 90 percent of the total cost of a typical software project [12].

This high cost could potentially be greatly reduced by providing automatic or semi-automatic solutions to increase their understandability, adaptability and extensibility to avoid bad-practices. As a result, there has been much research focusing on the study of bad design practices, also called code-smells, defects, anti-patterns or anomalies [13], [14], [15], [25] in the literature. Although these bad practices are sometimes unavoidable, they should be in general prevented by the development teams and removed from their code base as early as possible. In fact, detecting and removing these code-smells help developers to easily understand source code [13]. In this work, we focus on the detection of code-smells.

The vast majority of existing work in code-smells detection relies on declarative rule specification [15], [17], [18], [19], [20]. 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 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 [13]. When consensus exists, the same symptom could be associated to many code-smells types, which may compromise the precise identification of code-smell types.

To address these issues, we proposed an approach based on the use of genetic programming (GP) to generate detection rules from examples of code-smells using structural metrics [21]. However, the quality of the generated rules depends on the coverage of the different suspicious behaviors of code-smells, and it is difficult to ensure such coverage. Thus, there are still some uncertainties regarding the detected code-smells due to the difficulty to evaluate the coverage of the base of code-smell examples. In another recent work, we proposed an approach based on an artificial immune system metaphor to detect code-smells by deviation with examples of good code-practices and well-designed systems [22]. Some of the detected code fragments that are different from well-designed code were not code-smells but just new good-practice behavior. Thus, we believe that an efficient approach will be to combine both detection algorithms to find better consensus when detecting code-smells.

We propose in this paper to consider code-smells detection as a distributed optimization problem. The idea is that

- *W. Kessentini, H. Sahraoui, and A. Ouni are with the Department of Computer Science, University of Montreal, Montreal, Quebec, Canada. E-mail: {kessentw, sahraouh, ouniali}@iro.umontreal.ca.*
- *M. Kessentini and S. Bechikh are with the Department of Computer Science, University of Michigan, Dearborn, MI. E-mail: {marouane, slim}@umich.edu.*

Manuscript received 23 Apr. 2013; revised 3 Apr. 2014; accepted 11 June 2014. Date of publication 15 June 2014; date of current version 18 Sept. 2014. Recommended for acceptance by H.C. Gall.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2014.2331057

different methods are combined in parallel during the optimization process to find a consensus regarding the detection of code-smells. To this end, we used Parallel Evolutionary algorithms [2] where many evolutionary algorithms (EAs) with different adaptations (fitness functions, solution representations, and change operators) are executed, in a parallel cooperative manner, to solve a common goal which is the detection of code-smells. We believe that a P-EA approach is suitable to our problem because it allows us to combine the different levels of expertise (algorithms) to detect code-smells. We show how this combination can be formulated as cooperation between many populations during the search process. In P-EA, many populations' individuals evolve simultaneously. Indeed, a first population generates a set of detection rules using genetic programming [11] and simultaneously a second population tries to detect code-smells using a deviation from examples of well-designed codes [22]. Both populations are executed, on the same system to evaluate, and the quality of solutions is updated based on the consensus found, i.e.: intersection between the detection results of both populations. The best detection results will be the code-smells detected by the majority of algorithms. The P-EA approach does not consist on just executing the two search-based algorithms in parallel but to build a consensus between them to classify the detected candidates based on several interactions in the fitness function level.

We implemented our P-EA approach and evaluated it on nine systems using an existing benchmark. We report the results on the effectiveness and efficiency of our approach, compared to a random search (RS) and to different existing single population-based approaches [21], [22]. The statistical analysis of our results indicates that the P-EA approach has great promise; P-EA significantly outperforms random search, two single population-based approaches and two code-smells detection techniques (not based on meta-heuristics search) [16], [49] in terms of precision and recall based on existing benchmarks [16], [17], [23].

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

- A novel formulation of the code-smells detection as a distributed optimization problem. To the best of our knowledge and based on recent search-based software engineering (SBSE) surveys [24], this is the first work that uses parallel cooperative evolutionary algorithms to solve code-smells detection problem where various EAs, with different adaption schemes, are executed in parallel.
- An empirical evaluation to compare the implementation of our cooperative P-EA approach with random search, existing single population-based approaches [23], [24] and two code-smells detection techniques not based on meta-heuristics search [16], [49]. The statistical analysis of the obtained results provides evidence to support the claim that cooperative P-EA is more efficient and effective than state of the art refactoring solution on a benchmark of nine large open source systems.

The remainder of this paper is structured as follows: Section 2 presents the relevant background and the motivations for the presented work; Section 3 describes the cooperative parallel search scheme; the adaption of P-EA to code-smells detection is presented in Section 4; an evaluation of the algorithm is explained in Section 5 and its results are discussed in Section 6; different threats to validity are discussed in Section 7; Section 8 is dedicated to related work. Finally, concluding remarks and future work are provided in Section 9.

2 Software Refactoring: Open Problems

In this section, we first provide the necessary background of detecting code-smells and discuss the challenges and open problems that are addressed by our proposal.

2.1 Background

2.1.1 Code-Smells

Code-smells, also called design anomalies or design defects, refer to design situations that adversely affect the software maintenance [25]. As stated by [13], bad-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. Different types of code-smells, presenting a variety of symptoms, have been studied in the intent of facilitating their detection [13] and suggesting improvement solutions. In [25], Fowler et al. define 22 sets of symptoms of code smells. These include large classes, feature envy, long parameter lists (LPLs), and lazy classes (LCs). Each code-smell type is accompanied by refactoring suggestions to remove it. Brown et al. [13] define another category of code-smells that are documented in the literature, and named anti-patterns. In our approach, we focus on the eight 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.
- *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 code produced by non-experienced object-oriented developers.
- *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 stores data.
- *Lazy class*. A class that isn't doing enough to pay for itself.
- *Long parameter list*. Methods with numerous parameters are a challenge to maintain, especially if most of them share the same data-type.
- *Shotgun surgery (SS)*. It occurs when a method has a large number of external operations calling it, and these operations are spread over a significant number of different classes. As a result, the impact of a change in this method will be large and widespread.

We choose these code-smell types in our experiments because they are the most frequent and hard to detect and fix based on a recent empirical study [16], [18], [20], [25].

2.1.2 Code-Smells Detection

The code-smells' detection process consists in finding code fragments that violate structure 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 [26]. This follows a long tradition of using software metrics to evaluate the quality of the design including the detection of code-smells [14], [15]. The most widely-used metrics are the ones defined by Chidamber and Kemerer [26]. These metrics include: (1) Depth of Inheritance Tree (DIT), (2) Weighted Methods per Class (WMC), and (3) Coupling Between Objects (CBO). In this paper, we use variations of these metrics and adaptations of procedural ones as well, e.g. the number of lines of code in a class (LOCCLASS), number of lines of code in a method (LOCMETHOD), number of attributes in a class (NAD), number of methods (NMD), lack of cohesion in methods (LCOM5), number of accessors (NACC), and number of private fields (NPRIVFIELD).

2.2 Challenges and Open Problems

In the following, we introduce some code-smells' detection issues and challenges. Later, in Section 6, we discuss these issues in more detail with respect to our approach.

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 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.

Finally, detecting dozens of code-smells occurrences in a system is not always helpful except if the list of code-smells is sorted by priority. In addition to the presence of false positives that may create a rejection reaction from development teams, the process of using the detected lists, understanding the code-smell candidates, selecting the true positives, and correcting them is long, expensive, and not always profitable.

3 The Proposed Cooperative Parallel Model Scheme

Nowadays, real life optimization problems are more and more complex. Consequently, their resource requirements are ever increasing. Optimization problems are often hard and expensive from a CPU time and/or memory viewpoint [1]. The use of metaheuristics, such as Evolutionary Algorithms and Particle Swarm Optimization (PSO), allows reducing the computational complexity of the search process. However, the latter remains computationally costly in different application domains where the objective functions and the constraints are resource intensive and the size of the search space is huge. In addition, to the problem of complexity, we find today resource-expensive search methods such as hybrid metaheuristics and multi-objective ones. The rapid technology development in terms of processors, networks and data storage tools renders *parallel distributed computing* very interesting to use. This fact has motivated researchers to more focus on designing and implementing parallel metaheuristics, mainly P-EAs, in order to solve more complex optimization problems [2].

There are several motivations behind the use of parallel and distributed computing for the design and implementation of P-EAs. Firstly, parallelization allows speeding up the search process by reducing the search time. This is very interesting in time-dependent and interactive resolution methods. Second, the quality of the obtained solutions may be significantly improved. In fact, cooperative P-EAs have been demonstrated to explore the fitness landscape more efficiently on different problems such as the code-smell identification problem [3]. This is realized by portioning the search space and then exchanging information between the different search methods, which allows examining the search space more efficiently. Thirdly, the use of different metaheuristics (e.g., EAs) simultaneously in solving a given problem reduces the sensitivity to the parameter values. Indeed, each search method would be launched with a particular parameter value set which is different from the others' ones. Hence, the search process would work according to different parameter value sets which may augment the *accuracy* of the obtained results. Finally, P-EAs allow tackling the scalability. Several problems actually involve a very large number of decision variables (called large-scale problems [4]), a high number of objectives (called many-objective problems ([5]), a large number of constraints (called highly-constrained problems [6]), etc. These types of problems are very computationally costly. P-EAs can represent one possible remedy to tackle such problems.

Different models exist for designing P-EAs. According to Talbi [7], these models follow the following three hierarchical levels:

- *Algorithmic level*. In this parallel model, independent or cooperating self-contained EAs are used. It can be seen as a *problem-independent inter-algorithm parallelization*. If the different EAs are independent, the

search will be equivalent to the sequential execution of the EAs in terms of the quality of solutions. However, the cooperative model will alter the behavior of the EAs and enable the improvement of the quality of solutions.

- *Iteration level*. In this model, EAs' iterations are parallelized. It is *a problem-independent intra-algorithm parallelization*. The EA's behavior is not altered. The main goal is to speed up the algorithm by reducing the search time since the iteration cycle of metaheuristics, on large neighborhoods for trajectory-based metaheuristics or large populations for population-based metaheuristics (e.g., EAs), requires a large amount of computational resources.
- *Solution level*. In this model, the parallelization process handles a single solution from the search space. It is a *problem-dependent intra-algorithm parallelization*. In general, evaluating the objective function(s) or constraints for a particular generated solution is almost always the *most costly* operation in EAs. In this model, the EA's behavior is not altered. The goal is mainly the speed up of the search.

In our proposal, we use a parallelization at the solution level without interchanging solutions between the considered search algorithms. In fact, in each generation, the fitness values of the best solutions are updated based on an intersection score that is detailed later.

Several research questions should be answered when designing P-EAs [8]. The most important ones are the following:

- ***What*** *information to exchange between EAs?* In a parallel evolutionary model, EAs communicate between themselves. The message content could be composed of: (1) *elite solutions* that have been discovered during the search, e.g., the current iteration best solution, the locally best solutions, the global best solution, the neighborhood best solution, the most diversified solutions; and (2) *search memory* of the corresponding metaheuristic such as the pheromone trails for Ant Colony Optimization (ACO) and the probability model for Estimation of Distribution Algorithms (EDAs). In our parallel model, there is an implicit exchange of information without interchanging solutions. Indeed, the elite solutions' fitness values are updated based on the intersection score. Consequently, the information exchange could be seen as implicit since there is no solution migration between the search algorithms.
- ***How*** *to integrate the received information?* The exchanged information is generally used to update the elite solution(s) of the recipient EA. Different replacement strategies may be applied to the local population by using the set of received solutions. For example, an elitist replacement strategy will integrate the obtained k solutions by replacing the k worst solutions of the local population. The implicit information exchange based on the consensus score will update the elite solutions' fitness values. In this way, some elite solutions will be emphasized w.r.t the parallel model and some others will be discouraged to remain in the race; thereby pushing the search algorithms to update their convergence process behavior based on the consensus score.
- ***Where*** *to send the useful information?* Each parallel model is characterized by a particular communication exchange topology. The latter indicates for each EA its neighboring ones regarding the information exchange. In this way, the model will be able to detect the source/destination algorithm of the useful information. Among the most-used topologies, we find the ring, the mesh, the hypercube and the complete graph. Our parallel model topology could be seen as a ring where the search considered search algorithms communicates through the fitness update module.
- ***When*** *to exchange information?* The information exchange moment between EAs could be decided *periodically*, *probabilistically* or *adaptively*. When using a periodic strategy, a change occurs in each algorithm after a fixed number of iterations/number of evaluations. This type of communication is called *synchronous*. When using a probabilistically strategy, an exchange occurs after each iteration according to given appearance probability. The adaptive strategy exploits information issued from the search process, such as the solution quality, to decide whether to launch an exchange or not. We use a synchronous communication between the two search algorithms since we update the fitness values of elite solutions periodically each generation. However, other strategies could be tested.

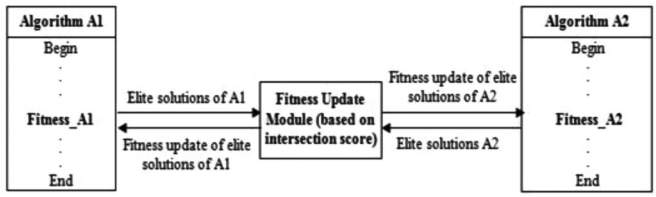


Fig. 1 The proposed cooperative parallel model scheme.

Although parallel distributed EAs' are still a challenging research field, several contributions are mature enough today to be exploited within the SBSE framework. Since the most studied and known models are based on EAs [9], the scope of this paper is to illustrate one of the first attempts of using P-EAs to solve software engineering problems, and particularly the code-smells detection one. Indeed, to the best of our knowledge and based on recent surveys [24], [47], this work represents the first research contribution to solve the code smells detection problem using P-EAs.

The parallelization proposed in this work intervenes in the solution level since it affects the solution evaluation module. Consequently, it could be seen as a *problem-dependent intra-algorithm* parallelization. In fact, in each generation of the parallel process, the top 5 percent elite solutions, i.e., those with the highest fitness values, are updated according to an intersection component. The update operation role is to penalize solutions that do not perform well according to the parallel model and to favor good solutions w.r.t. the parallelization. For the problem considered in this work, code-smells detection,

we use a simple *bi-algorithm parallel* model illustrated by Fig. 1. We see, from this figure, that the fitness of elite solutions of EA1 is updated based on the fitness values of elite solutions of EA2 and vice versa. The main goal of the fitness-update operation is alleviating the encountered challenges of the code-smell detection, which are discussed in the previous section. Indeed, the parallel model goal is *to circumvent the absence* of a well-defined consensus concerning the detection of code-smells (symptoms, threshold values, etc.). The next section describes in details our adaptation of P-EAs to our code-smells detection problem.

4 Parallel Evolutionary Algorithms Adaptation

This section shows how parallel meta-heuristic search can be adapted to our problem of code-smells detection. In order to ease the understanding of this formulation, we first describe the pseudo-code of our adaptation. Then, we present the solution structure, the objective functions to optimize, and finally, the used change operators.

The first used evolutionary algorithm is based on genetic programming [27] to generate code-smells detection rules. The second evolutionary algorithm executed in parallel is genetic algorithm GA [11] that generates detectors (code-smells examples) from well-designed code examples. Both algorithms are powerful metaheuristic search optimization methods inspired by the Darwinian theory of evolution [27]. The basic idea of both algorithms is to explore the search space by making a population of candidate solutions, also called individuals, evolve towards a *"good"* solution of a specific problem. To evaluate the solutions, the fitness function in both algorithms has two components. For the first component of the fitness function, GP evaluates the detection rules based on the coverage of code-smells examples (input) and GA evaluates the detectors by calculating the deviance from well-designed code (input) using global and local alignment techniques [35]. Then, a set of best solutions are selected from both algorithms in each iteration. Both algorithms interact with each other using the second component of the fitness function called *intersection_function* where a new system, different from the ones used to produce the inputs, is evaluated using these solutions in order to maximize the intersection between the sets of detected code-smells by each solution. Thus, some solutions will be penalized due to the absence of consensus and others will be favored. In the initialization of the P-EA algorithm, one system from the base of examples is selected randomly to calculate the intersection function and removed from the base of examples used by the GP algorithm. Thus, P-EA is to be run for the system selected from the base of examples that needs to have code-smell(s) detected, as the intersections will be determined for this particular system and the system to evaluate is not considered in the training set. The best rules and detectors can be used to evaluate any new system without the need to execute again P-EA. The developers should execute P-EA again only in the case that major revisions are introduced to the base of examples.

The result of both algorithms executed in parallel is the fittest individual produced along all generations for each one. In our case, the P-EA generates the best detection rules and the best detectors. These rules and detectors can be executed by developers to detect code-smells on new systems.

```
Input: Set of quality metrics M.
Input: Set of systems S evaluated manually (defects examples).
Input: New system A.
Output: Detection rules.
1: I1:= rules(M, Defect_Type);
2: P1:= set_of(I1);
3: initial_populationGP(P1, Max_size) ;
4: repeat
5:   for all I1 ∈ P1 do
6:          detected_defects _GP(S) := execute_rules(I1, S);
7:          fitness(I1) := compare(detected_defects,   defects_examples);
8:   end for
9:   best_sol_P1 := select(P1, number_best_solutions);
10: send (best_sol_P1);
11: best_sol_P2 := receive(best_sol_P2);
12:     for all I1 ∈best_sol_P1 do
13:         detected defects _GP(A) := execute_rules(I1, A);
14:         fitness_intersection (I1) := Max_intersection(detected defects _GP(A, I1) ∩
                                        detected_defects _GA(A, best_sol_P2));
15:             fitness(I1) := updatefitness (I1, fitness_intersection);
16: end for
17: best_solution_P1 := best_fitness(P1);
18:     P1 := generate_new_population(P1);
19: it:=it+1;
20: until it=max_it
21: return best_solution_rules
```

(a)

```
Input: Set of well-designed code examples GE.
Input: New system A
Output: Detectors.
1: I2 := detectors(GE);
2: P2= set_of(I2) ;
3: initial_populationGA(P2, Max_size) ;
4: repeat
5:   for all I2 ∈P2 do
6:          fitness(I2) := distance(detectors_I2,   GE);
7:   end for
8:   best_sol_P2 := select(P2, number_solutions);
9:   send(best_sol_P2);
10: best_sol_P1 := receive(best_sol_P1);
11:     for all I2 ∈best_sol_P2 do
12:         detected_defects _GA(A) := execute_detectors(I2, A);
13:         fitness_intersection (I2) := Max_intersection(detected defects _GA(A, I2) ∩
                                        detected_defects _GP(A, best_sol_P1));
14:             fitness(I2) := updatefitness (I2, fitness_intersection);
15: end for
16: best_solution_P2 := best_fitness(P2);
17:     P2 := generate_new_population(P2);
18: it:=it+1;
19: until it=max_it
20: return best_solution_detectors
```

(b)

Fig. 2. High-level pseudo-code for P-EA adaptation to our problem: (a) GP is executed to generate detection rules from code-smell examples; (b) GA is executed in parallel to generate detectors from well-designed code examples. Both algorithms interact with each other using *"fitness_intersection"* function in each interaction.

A high-level view of our P-EA approach to the code-smells detection problem is introduced by Fig. 2 where GA and GP algorithms are executed in parallel and interact using a fitness function in each generation.

Lines 1-3 construct an initial GP and GA populations, which are sets of individuals that stand for possible solutions representing detection rules (metrics combination) for GP and detectors (artificial code that represents pseudo code-smell examples) for GA. Lines 4-15 encode the main GP and GA loops, which explores the search space and constructs new individuals by combining metrics for GP to generate rules and code elements to generate detectors for GA.

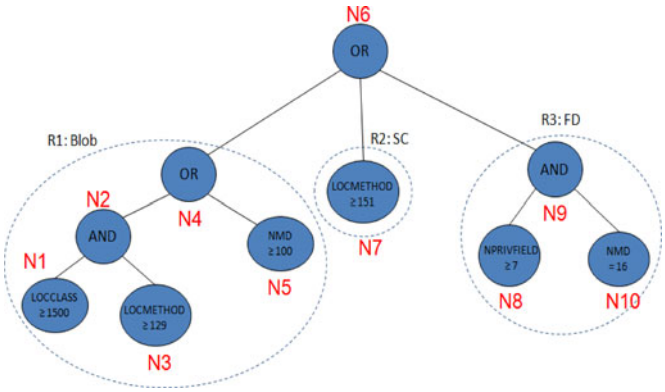


Fig .3. Solution representation: tree (GP) to generate rules.

$Detector_{32}$

C	G	A	A	M	P	P	M

$Detector_{32}$
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);

Fig. 4. Solution representation: vector (GA) to generate detectors.

In each iteration, we evaluate the quality of each individual in the population for each algorithm (lines 6 and 13) as described before. Then, the best solution for each algorithm is saved and a new population of individuals is generated by iteratively selecting pairs of parent individuals from population p and applying the crossover operator to them. We include both the parent and child variants in the new population *pop*. Then, we apply the mutation operator, with a probability score, for both parent and child to ensure the solution diversity; this produces the population for the next generation. When applying change operators, no individuals are exchanged between the parallel GA/GP. Both algorithms terminate when the termination criterion (maximum iteration number) is met, and returns the best set of rules and detectors. Finally, developers can use the best rules and detectors to detect code-smells on any new system to evaluate. The tool ranks the detected code-smells based on an intersection score that will be discussed in Section 5.

In the following, we describe the three main steps of adaptation of both GP and GA algorithms to our problem.

4.1 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. 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 solutions (rules) corresponds to a logic program that is represented as a forest of AND-OR trees. A solution is a set of detection rules where each rule detects a specific type of code-smell. Thus, every solution has a number of rules equivalent to the number of types of code-smells to detect. For example, consider the following logic program described in Fig. 3:

R1: *IF (LOCCLASS $\geq$ 1500 AND LOCMETHOD $\geq$ 129) OR (NMD $\geq$ 100) THEN code-smell = blob*
R2: *IF (LOCMETHOD $\geq$ 151) THEN code-smell = functional decomposition*
R3: *IF (NPRIVFIELD $\geq$ 7 AND NMD = 16) THEN code-smell = spaghetti code*

For GA, detectors represent generated artificial code fragments composed by code elements. Thus, detectors 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 *CGAAMPPM* corresponds to a class with a generalization link, containing two attributes and two methods. 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.

To generate an initial population for both algorithms, 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-smells 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. Fig. 4 shows an example of a generated detector composed by 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.).

4.2 Solution Evaluation

The encoding of an individual should be formalized as a mathematical function called the "fitness function". The

Detection results

Class	Blob	Functional decomposition	Spaghetti code
Student		X	
Person		X	
University		X	
Course	X		
Classroom			X
Administration	X		

Code-smells in the base of examples

Class	Blob	Functional decomposition	Spaghetti code
Person		X	
Classroom	X		
Professor		X	

$$f_1 = \frac{\frac{1}{3}+\frac{1}{6}}{2} = 0.25$$

Fig. 5. Coverage of the base of code-smell examples.

fitness function quantifies the quality of the proposed detection rules and detectors. The goal is to define efficient and simple fitness functions in order to reduce the computational cost. For our GP adaptation, to evaluate detection-rules solutions the fitness function is based on: (1) maximizing the coverage of the base of code-smell examples and (2) maximizing the consensus, using an intersection function, with the GA, executed in parallel, regarding detected code-smells on a new system A. For our GA adaptation, to evaluate generated detectors the fitness function is based on: (1) a dissimilarity score, to maximize, between detectors and different reference code fragments while minimizing the overlap between detectors (to maximize diversity) and (2) maximizing the intersection function. In the following, we detail all these functions.

Coverage of the base of code-smell examples (used by GP). This objective function checks to maximize the number of detected code-smells in comparison to the expected ones in the base of examples. In this context, we define the objective function of a particular solution *S*, normalized in the range [0,1] as follows:

$$f\_\text{cov}\,erage(S) = \frac{\frac{\sum_{i=1}^{p} a_i(S)}{t} + \frac{\sum_{i=1}^{p} a_i(S)}{p}}{2},$$

where p is the number of detected code-smells after executing the solution (detection rules) on systems of the base of code-smell examples, t is the number of expected code-smells to detect in the base of examples and $a_i(S)$ is the ith component of S such that:

$$a_i(S) = \begin{cases} 1 & \text{if the ith detected code smell exists} \\ & \text{in the base of examples} \\ 0 & \text{otherwise} \end{cases}$$

Fig. 5 illustrates the above-described function where we consider a base of examples containing one system evaluated manually. The true code-smells of this system are described in the base of examples' table. The classes detected after executing the solution generating the rules *R*1, *R*2 and *R*3 are described in the detection result table. Thus, only one class corresponds to a true code-smell (Person). Classroom is a code-smell, but the type is wrong and Professor is not a code-smell. The fitness function has the value 0.25 as illustrated in the figure.

The base of examples used by the GP contains only examples of code smells and not examples of good code. In fact, an example of BLOB could be at the same time an example of non-spaghetti code. However, if we consider that there is only one type of code smell, then a rule that simply classifies everything as being code smell would obtain very high fitness if the base of examples does not contain any example of good code.

Detector evaluation (used by GA). This section describes how a set of detectors is produced starting from the reference code examples. The idea is to produce a set of detectors that best covers the possible deviations from the reference code. As the set of possible deviations is very large, its coverage may require a huge number of detectors, which is infeasible in practice. For example, pure random generation was shown to be infeasible in [30] for performance reasons. We, therefore, consider the detector generation as a search problem. A generation algorithm should seek to optimize the following two objectives:

1) Maximize the generality of the detector by minimizing the similarity with the reference code examples.
2) Minimize the overlap (similarity) between detectors.

These two objectives define the cost function that evaluates the quality of a solution and, then guides the search. The cost of a solution *D* (set of detectors) is evaluated as the average costs of the included detectors. We derive the cost of a detector d_i as a weighted average between the scores of respectively, the lack of generality and the overlap. Formally,

$$\cos t(d_i) = \frac{LG(d_i) + O(d_i)}{2}.$$

Here, we give equal weight to both scores. The lack of generality is measured by a matching score $LG(d_i)$ between the predicate sequence of a detector d_i and those of all the classes s_j in the reference code (call it *RC*). It is defined as the average value of the alignment [22] scores $Sim(d_i, s_j)$ between d_i and classes s_j in RC. Formally,

$$LG_{d_i} = \frac{\sum_{s_j \in S} Sim(d_i, s_j)}{|S|}.$$

Similarly, the overlap O_i, is measured by the average value of the individual $Sim(d_i, d_j)$ between the detector d_i and all the other detectors d_j in the solution *D*. Formally,

$$O_{d_i} = 1 - \frac{\sum_{d_j, j \neq i} Sim(d_i, d_j)}{|D|}.$$

To calculate the similarity *Sim()* between two code fragments, we adapted the Needleman-Wunsch alignment algorithm [28] to our context. It is a dynamic programing algorithm used in bioinformatics to efficiently find similar regions between two sequences of *DNA*, *RNA* or protein [31]. Because the Needleman-Wunsch algorithm finds the optimal alignment of the entire sequence of both proteins, it is a global alignment technique, and cannot be used to find local regions of high similarity. Global Alignment assumes that the two proteins are basically similar over the entire length of one another. The alignment attempts to match them to each other from end to end, even though parts of

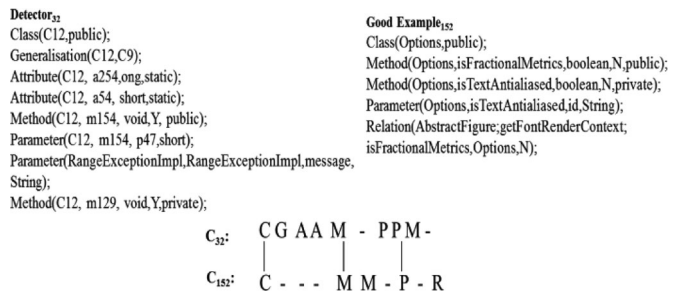


Fig. 6. Global alignments of two strings (code-fragments).

the alignment are not very convincing. An example of the algorithm is presented in Fig. 6. To adapt the global alignment to our code-smells detection problem, we represented the source code as textual strings as described in the solution representation section. We used textual string representation since we are interested more to the structure of the source code. Thus, we do not need to use the syntax tree representation. In addition, the syntax tree representation cannot be used to compare two code-fragments using the alignment algorithms used in our adaption. The textual representation can reduce the computational cost required to compare the different code-fragments.

The Needleman-Wunsch alignment algorithm has been used in different work to detect code clone [32], [33] with interesting results. This algorithm can efficiently compare two code fragments quickly. This represents one of the main motivations to choose a global alignment algorithm since we need to define an efficient fitness function with the lowest computational complexity.

The Needleman-Wunsch global alignment algorithm [28] is described recursively. When aligning two sequences $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_m)$, each position $s_{i,j}$ in the matrix corresponds to the best score of alignment considering the previously aligned elements of the sequences. The algorithm can introduce gaps (represented by "−") to improve the matching of subsequences.

$$s_{i,j} = Max \begin{cases} s_{i-1,j} - g//\text{insert gap for b}_\text{j} \\ s_{i,j-1} - g//\text{insert gap for a}_\text{i} \\ s_{i-1,j-1} + sim_{i,j}//\text{match} \end{cases},$$

where $s_{i,0} = g * i$ and $s_{0,j} = g * j$ when aligning two sequences $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_m)$.

At any given point, the algorithm considers two possibilities. First, it considers the case when a gap should be inserted. When a gap is inserted for either *a* or *b*, the algorithm applies a penalty of *g*. Second, it tries to match predicates. The similarity function $sim_{i,j}$ returns the reward or cost of matching a_i to b_j. The final similarity is contained in $s_{n,m}$.

Our adaptation of the algorithm is straightforward. We define the gap penalty *g* and the similarity function to match individual predicates (*sim*). We do not seek perfect matches in terms of number of predicates. A class with *four* methods is not necessarily different from one with *six* methods if the methods are similar. To eliminate the sensitivity of the algorithm to size, we set the gap penalty to 0.

We define a predicate-specific function to measure the similarity. First, if the types differ, the similarity is *0*. As we

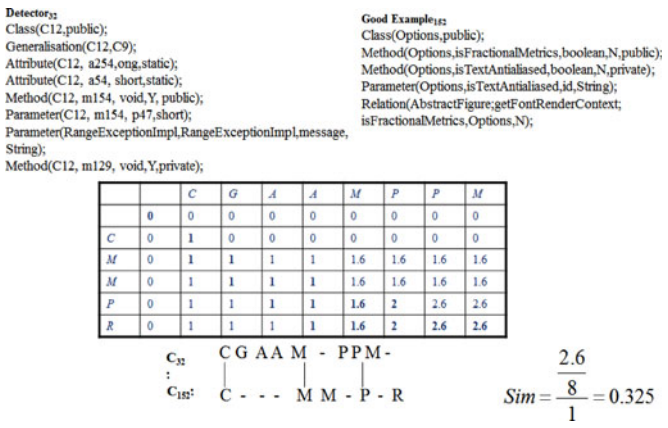


Fig. 7. Global alignment illustration.

manipulate sequences of complex predicates and not strings, $sim_{i,j}$ is defined as a predicate-matching function, PM_{ij}. PM_{ij} measures the similarity with respect to the elements of the predicates associated to a_i and b_j. This similarity is the ratio of common parameters in both predicates:

$$PM_{ij} = \frac{\forall_{p \in a_i, q \in b_i} \cap (p, q)}{\max(|a_i|, |b_j|)},$$

where a_i and b_j are treated as sets of predicates. The equivalence between predicate parameters depends on each type of parameter. For visibility and element types, it means equality. Specific names are not considered. Instead, they are used to indicate a common reference by other predicates. For example, if a class defines an attribute and its related getter method, they will both share the same class name.

To illustrate an example for the alignment algorithm, let us consider the detector (d32) generated by GA, described previously and a reference code fragment (c152) as described in Fig. 7. The goal is to evaluate the similarity, to maximize, between these two code fragments: the detector generated by GA and an example of a reference code fragment. The code fragments are sequentially numbered. According to the coding mentioned previously, the predicate sequence for d32 is CGAAMPPM and the one of C152 is CMMPR. The alignment algorithm finds the best alignment sequence as shown in Fig. 7. There are three matched predicates between d32 and c152: one class, one method, and one method parameter. If we consider the second matched predicates Method(C12, m154, void, Y, public) from d32 and Method(Options, isFractionalMetrics, Boolean, N, public) from c152. The predicates have two common parameters out of possible five ones. The resulting similarity is consequently 40 percent. We normalize this absolute similarity measure, $s_{n,m}$, by the maximum number of predicates to produce our overall similarity measure *Sim(A,B)* between two code fragments (classes):

$$Sim(A, B) = \frac{s_{n,m}}{Max(n, m)},$$

where *A* and *B* represent the two code fragments (classes) to compare, *n* is the number of predicates (code elements) in *A* and *m* is the number of predicates in *B*. Thus, the *Sim* measure is used to evaluate the quality of the generated detectors where the objective to maximize the distance the

distance between the generated detectors and the reference code examples.

Intersection score (used by both algorithms). In a real-world scenario, there is no consensus about the definition of code-smells since developers can have diverged opinions about the detection of code-smells then they can converge to a final decision. In general, the consensus is built based on the majority of votes that correspond in our case to the intersection between different class candidates. Furthermore, only 50% of our defined fitness function is based on maximizing the consensus (intersection). In addition, the consideration of the union of GA and GP solutions may increase the recall and not the precision. A huge list of candidates will be proposed to the designer in this case and it is time consuming for him to validate the results to find code-smells. Larger intersection will improve both the precision and recall scores. In fact, if the classes are detected by both algorithms then there is a high probability that they are actual code-smells. To maximize the intersection there are two alternatives: 1) most of the defects are detected by algorithm A (and not by algorithm B) at iteration *i* will be detected in the future by algorithm B; or 2) most of the defects detected by algorithm A (and not by algorithm B) at iteration *i* will not be detected anymore in the future by algorithm A. In our adaption, since GA can detect the defects based on a risk score thus in the case that the risk of a defect is high and it is not detected by GP then this defect will be detected in the next iterations of GP because it is the only way to maximize the overall fitness functions of GP and GA (including the intersection). In the case that the risk score affected to a code-smell is low then GA will probably not detect this code-smell in the next iteration because this is the only way to maximize the overall fitness functions of GA and GP (including the intersection).

A set of best solutions are selected from both algorithms, in each iteration, and then executed on a new system A to evaluate. A matrix is constructed where rows are composed by best solutions of GP $\{SGP_i\}$, columns are composed by best solution of GA $\{SGA_j\}$ and each case (SGP_i, SGA_j) is defined as

$$f_{\text{intersection}}(SGP_i, SGA_j) = \frac{PR}{2}, where$$
$$PR = \frac{\left|\{\text{code-smells detected by SGP}_i \text{ on A}\} \cap \{\text{code-smells detected by SGA}_j \text{ on A}\}\right|}{\text{Max}(\#\text{code-smells detected by SGP}_i \text{on A}, \#\text{code-smells detected by SGA}_j \text{on A})} + \frac{\left|\{\text{code-smells detected by SGP}_i \text{on A}\} \cap \{\text{code-smells detected by SGA}_j \text{on A}\}\right|}{\#\text{actual code-smells in A}}$$

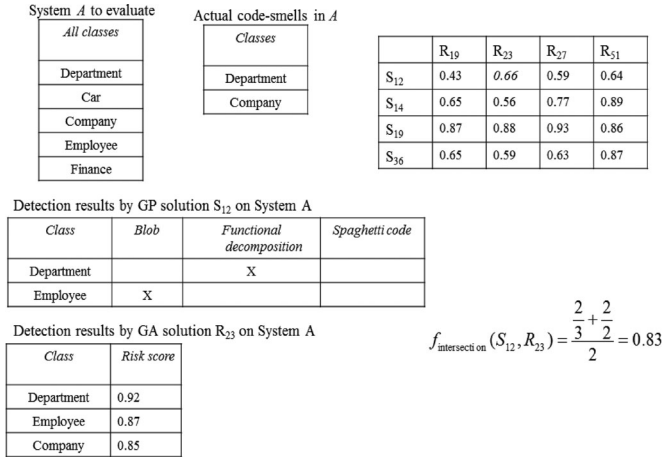
Then, the intersection score for each solution is defined as $f_{\text{intersection}}(SGP_i) = Min\left(f_{\text{intersection}}(SGP_i, SGA_{\forall j})\right)$ and $f_{\text{intersection}}(SGA_j) = Min\left(f_{\text{intersection}}(SGP_{\forall i}, SGA_j)\right)$.

To sum-up, the fitness function of the GP algorithm for a solution S_i is defined as

$$fitness\_GP(S_i) = \frac{f\_\text{cover}age(S_i) + f_{\text{intersection}}(S_i)}{2}$$

where

$$f_{\text{intersection}}(S_i) = \begin{cases} 0, \text{if the solution S is not selected among best solutions} \\ f_{\text{intersection}}(SGP_i), \text{otherwise} \end{cases}$$



	R_{19}	R_{23}	R_{27}	R_{51}
S_{12}	0.43	*0.66*	0.59	0.64
S_{14}	0.65	0.56	0.77	0.89
S_{19}	0.87	0.88	0.93	0.86
S_{36}	0.65	0.59	0.63	0.87

Fig. 8. Illustration of the intersection function.

For GA, the best detectors (solution) are used to detect code-smells. The system to evaluate is compared using the alignment algorithm to the obtained detectors. The risk of being a code-smell, associated to a code fragment e_i is defined as the average value of the alignment scores $Sim(e_i, d_j)$ obtained by comparing e_i to respectively all the detectors of a set D. Formally,

$$risk_{e_i} = \frac{\sum_{d_j \in D} Sim(e_i, d_j)}{|D|}.$$

The code fragments can then be ranked according to their risks to be inspected by the maintainers. In our adaptation, we consider a code fragment to be a code-smell only if the risk is more than 75 percent. Thus, the fitness function of the GA for a solution O_j is defined as

$$fitness\_GA(O_j) = \frac{\cos t(O_j) + f_{\text{intersection}}(O_j)}{2}$$

where

$$f_{\text{intersection}}(O_j) = \begin{cases} 0, \text{if the solution } O_j \text{is not selected among best solutions} \\ f_{\text{intersection}}(SGA_j), \text{otherwise} \end{cases}$$

Fig. 8 shows a new system A to evaluate. It contains 5 classes. Four best solutions generated by GP (S) and four best solutions are generated by GA are executed in parallel on the system A. Thus, a (4×4) matrix is generated to calculate the intersection between the detect code-smells as illustrated in Fig. 8. For example, two out of three are the same detected code-smells using solution S_{12} and R_{23}.

4.3 Evolutionary Operators

4.3.1 Selection

One of the most important steps in any EA is selection. There are two selection phases in EAs: (1) parent selection (also named mating pool selection) and (2) environmental selection (also named replacement) [27]. 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. Concerning 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 population P and Q into U ($|U| = |P| + |Q| = 2|P|$), and then the population P for the next generation will be composed by the $|P|$ fittest individuals from U. By doing this, we ensure that we do not encourage the survival of a worse individual over a better one. We can say that this environmental selection is elitist, which is a desired property in modern EAs [7].

4.3.2 Mutation

For GP, the mutation operator can be applied to a function node, or a terminal node. It starts by randomly selected 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 subtree are replaced by a new randomly generated subtree. For GA, The mutation operator consists of randomly changing a predicate (code element) in the generated predicates.

4.3.3 Crossover

For GP, two parent individuals are selected and a subtree is picked on each one. Then crossover swaps the nodes and their relative subtrees 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, the crossover operator allows to create two offspring o1 and o2 from the two selected parents p_1 and p_2. It is defined 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.

For instance, if $k = 3$ and $p_1 = \text{CAMMPPP}$ and $p_2 = \text{CMPRMPP}$, then $o_1 = \text{CAMRMPP}$ and $o_2 = \text{CMPMPPP}$.

Finally, when applying change operators, no individuals are exchanged between the parallel GA/GP.

5 Evaluation

In order to evaluate our approach for detecting code-smells using P-EA, we conducted a set of experiments based on different versions of real-world models extracted from large open source systems. Each experiment is repeated 51 times, and the obtained results are subsequently statistically analyzed with the aim to compare our proposal with some single population-based approaches [21], [22] in addition to random search and two existing detection techniques not based on meta-heuristic search [16], [49]. In this section, we start by presenting our research questions. Then, we describe and discuss the obtained results.

5.1 Research Questions and Objectives

The study was conducted to quantitatively assess the completeness and correctness of our code-smells detection approach when applied in real-world settings and to compare its performance with existing approaches. More specifically, we aimed at answering the following research questions (*RQs*):

- *RQ1.* To what extent can the proposed approach detect efficiently code-smells (in terms of correctness and completeness)?
- *RQ2.* What types of code-smells does it locate correctly?
- *RQ3.* To what extent does the cooperative parallel meta-heuristic approach performs better than the considered single-population ones?
- *RQ4.* How does P-EA perform compared to the existing code-smells detection approaches not based on the use of metaheuristic search?

To answer *RQ1*, we used an existing corpus [16], [17], [23] containing an extensive study of code-smells on different open-source systems: ApacheAnt[1], Xerces-J[2], GanttProject[3], Rhino[4], Log4J[5], Lucene[6], Nutch[7] and JFreeChart[8]. Our goal is to evaluate the correctness and the completeness of our P-EA code-smells detection approach. For RQ2, we investigated the type of code-smells that were found. For RQ3, we compared our results to those produced, over 51 runs, by existing single-population approaches [21], [22] in addition to random search. Further details about our experimental setting are discussed in the next subsection. 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 such as DECOR [17] and JDeodorant [49]. We selected both techniques because they can detect some of the code-smell types considered in this paper and they are widely used in the literature. Furthermore, both techniques are based on the use of quality metrics to detect code-smells and it is interesting to evaluate the ability of our proposal to better define the detection rules based on quality metrics. The current version of JDeodorant is implemented as an Eclipse plug-in that identifies some

1. http://ant.apache.org/.
2. http://xerces.apache.org/xerces-j/.
3. http://sourceforge.net/p/ganttproject/.
4. https://developer.mozilla.org/en-US/docs/R .
5. http://logging.apache.org/log4j/2.x/.
6. http://lucene.apache.org/.
7. http://nutch.apache.org/.
8. http://www.jfree.org/.

TABLE 1
Studied Systems

Systems	*Release*	*#Classes*	*#Smells*	*KLOC*
JFreeChart	v1.0.9	521	94	170
GanttProject	v1.10.2	245	72	41
ApacheAnt	v1.5.2	1024	172	255
ApacheAnt	v1.7.0	1839	169	327
Nutch	v1.1	207	79	39
Log4J	v1.2.1	189	68	31
Lucene	v1.4.3	154	41	33
Xerces-J	V2.7.0	991	114	238
Rhino	v1.7R1	305	82	57

types of design defects using quality metrics and then proposes a list of refactoring strategies to fix them. Moha et al. 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. These techniques are not dedicated to all the eight code-smell types that we considered in our experiments thus we performed the comparison using only some specific types of code-smells.

5.2 Setting

Our study considers the extensive evaluation of nine open-source Java analyzed in the literature. 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 1 reports the size in terms of classes of the analyzed systems. The table also reports the number of code-smells identified manually in the different systems. More than 800 code-smells have been identified manually. Indeed, in [17], [22], authors asked different groups of developers to analyze the libraries to tag instances of specific code-smells to validate their detection techniques. In our study, we verified the capacity of our approach to locate classes that correspond to instances of eight different types of code-smells: Blob, Spaghetti Code, Functional Decomposition, Feature Envy, Data Class, Lazy Class, Long Parameter List, Shotgun Surgery.

We selected these systems for our validation because they range from medium to large-sized open source projects 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.

For GP, one open source project is evaluated by using the remaining systems as a base of code-smells' examples to generate detection rules. For GA, JHotdraw [34] was chosen as an example of reference code because it contains very few known code-smells. In fact, previous work [48] 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. In general, after executing our P-EA technique the best detectors and detection rules are used to find code-smells in new systems. These code-smells are ranked using a severity score defined as

$$Severity(c_i) = \frac{Risk(c_i)+RulesDet(c_i)}{2},$$

where $Risk(c_i)$ represents the maximum global alignment distance between c_i and detectors, and $RulesDet(c_i)$ takes the value 1 if the class c_i is detected by the set of rules otherwise it takes 0. We consider a risk of 0.75 as an acceptable threshold value to consider a class as a code-smell. In our experiments, we performed a nine-fold cross validation to remove the system to evaluate from the base of examples. Thus, we repeated this process nine times and we executed our P-EA approach 51 times on every system to evaluate for the statistical tests.

To assess the accuracy of our approach, we compute two measures, precision and recall, 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. 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 remove the system to evaluate from the base of code-smell examples when executing our P-EA algorithm then precision and recall scores are calculated automatically based on a comparison between the detected code-smells and expected ones. We compared our results with existing single-population approaches [21], [22]. Therefore, we consider GP and GA as separate algorithms. In addition, we compared our P-EA approach to random search. We used precision and recall scores for all these comparisons over 51 runs. Since the used algorithms are stochastic optimizers, they may produce different results when applied to the same problem instance over different runs. To cope with this stochastic nature, the use of a rigorous statistical testing is essential to provide support to the conclusions derived from analyzing such data. Thus, we used the Wilcoxon rank sum test in a pairwise fashion [35] in order to detect significant performance differences between the algorithms under comparison. The Wilcoxon test allows testing the null hypothesis H0 that states that both algorithms' medians' values for a particular metric are not statistically different against H1 which states the opposite. The Wilcoxon test does not require that the data sets follow a normal distribution since it operates on values' ranks instead of operating on the values themselves [35]. Since we are comparing more

TABLE 2
Precision Median Values of PEA, GP, GA, and RS over 51 Independent Simulation Runs

Systems	PEA Precision (%)	GP Precision (%)	GA Precision (%)	RS Precision (%)
JFreeChart v1.0.9	84 (+++)	77 (+++)	68 (+++)	24 (+++)
Gantt v1.10.2	92 (+++)	81 (+++)	84 (+++)	32 (+++)
Lucene v1.4.3	87 (–+)	84 (–+)	77 (–+)	26 (+++)
Log4J v1.2.1	90 (–+)	86(–+)	83 (–+)	22 (+++)
Nutch v1.1	88 (-++)	82 (-++)	83 (+++)	26 (+++)
Xerces-J v2.7.0	93 (+++)	84 (+++)	87 (+++)	23 (+++)
Ant-Apache v1.5.2	91 (+++)	84 (+++)	82 (+++)	29 (+++)
Ant-Apache v1.7.0	86 (+++)	82 (+++)	81 (+++)	26 (+++)
Rhino v1.7R1	89 (+++)	81 (+++)	86 (+++)	34 (+++)

A "+" symbol at the ith position means that the algorithm precision median value is statistically different from the ith algorithm one. A "-" symbol at the ith position means the opposite (e.g., for Lucene v.1.4.3, GP precision is not statistically different from PEA and GA ones, however, it is statistically different from RS one).

TABLE 3
Recall Median Values of PEA, GP, GA, and RS over 51 Independent Simulation Runs

Systems	PEA Recall (%)	GP Recall (%)	GA Recall (%)	RS Recall (%)
JFreeChart v1.0.9	81 (+++)	64 (+++)	68 (+++)	21 (+++)
Gantt v1.10.2	90 (+++)	81 (+++)	87 (+++)	24 (+++)
Lucene v1.4.3	88 (–+)	77 (–+)	85 (–+)	22 (+++)
Log4J v1.2.1	83 (-++)	76 (-++)	77 (+++)	16 (+++)
Nutch v1.1	87 (+++)	86 (+++)	84 (+++)	17 (+++)
Xerces-J v2.7.0	91 (+++)	85 (+++)	82 (+++)	28 (+++)
Ant-Apache v1.5.2	86 (+++)	78 (+++)	84 (+++)	25 (+++)
Ant-Apache v1.7.0	84 (+++)	86 (+++)	82 (+++)	29 (+++)
Rhino v1.7R1	92 (+++)	84 (+++)	81 (+++)	33 (+++)

A "+" symbol at the ith position means that the algorithm precision median value is statistically different from the ith algorithm one. A "-" symbol at the ith position means the opposite (e.g., for Log4J v1.2.1, PEA recall is not statistically different from GP one, however it is statistically different from GA and RS ones).

than two different algorithms, we performed several pairwise comparisons based on Wilcoxon test to detect the statistical difference in terms of performance. To compare two algorithms based on a particular metric, we record the obtained metric's values for both algorithms over 51 runs. After that, we compute the metric's median value for each algorithm. Besides, we launch the Wilcoxon test with a 95 percent confidence level ($\alpha = 0.05$) on the recorded metric's values using the Wilcoxon MATLAB routine. If the returned p-value is less than 0.05 than, we reject H0 and we can state that one algorithm outperforms the other, otherwise we cannot say anything in terms of performance difference between the two algorithms. To compare our stochastic approach P-EA and the two deterministic methods DECOR and JDeodorant, we performed also the Wilcoxon test. All obtained results were statistically different against these deterministic methods. This observation is expected since each of DECOR and JDeodorant provides the same output over the 51 runs.

For our experiment, we generated 150 detectors from deviation with JHotDraw (about a quarter of the number of examples) with a maximum size of 256 characters. The same set of detectors was used on the different open source systems. For GP and GA, population size is fixed at 100 and the number of generations at 1,000. For P-EA, the population size is 100 and number of generations 500. In this way, all algorithms perform 100,000 evaluations (fair comparison). A maximum of 15 rules per solution and a set of 13 metrics are considered for GP [9], [27]. These standard parameters are widely used in the literature [9], [11]. We used the trial and error method to set the population size and the number of generations for each algorithm. This means that we have made several experiments using different values for these parameters. Following these experiments, we concluded that when using a population size of 100 for GA/GP, the fitness function becomes stabilized around the 1000th generation. The same was done for the parallel EA where we concluded that the fitness becomes stabilized from the 500th generation. For these reasons, the algorithms did not suffer from premature convergence; thereby the comparison is fair not only from the stopping criterion viewpoint but also from the parameter setting one.

For the variation operators, we used crossover probability of 0.9 and a mutation one of 0.5. We used a high mutation rate since we are employing an elitist schema for both GP and GA. In fact, as noted by Cohen [57], elitism may encourage premature convergence to occur. In order to avoid such a problem, in each generation, we emphasize the diversity of the population by means of the high mutation rate.

5.3 Results

Tables 2 and 3 summarize our findings. We only considered classes as a code-smell with a severity level that is greater than or equal 75 percent; around 5 percent of the classes in the system are detected as code-smells in the different systems. Overall, as described in Table 2, we were able to detect code-smells on the different systems with an average precision higher than 85 percent. For Gantt, Xerces, and Log4J, the precision is higher than for the other systems with more than 90 percent. This can be explained by the fact that these systems are smaller than others and contain a lower number of code-smells to detect. For Ant-Apache, the precision is also high (around

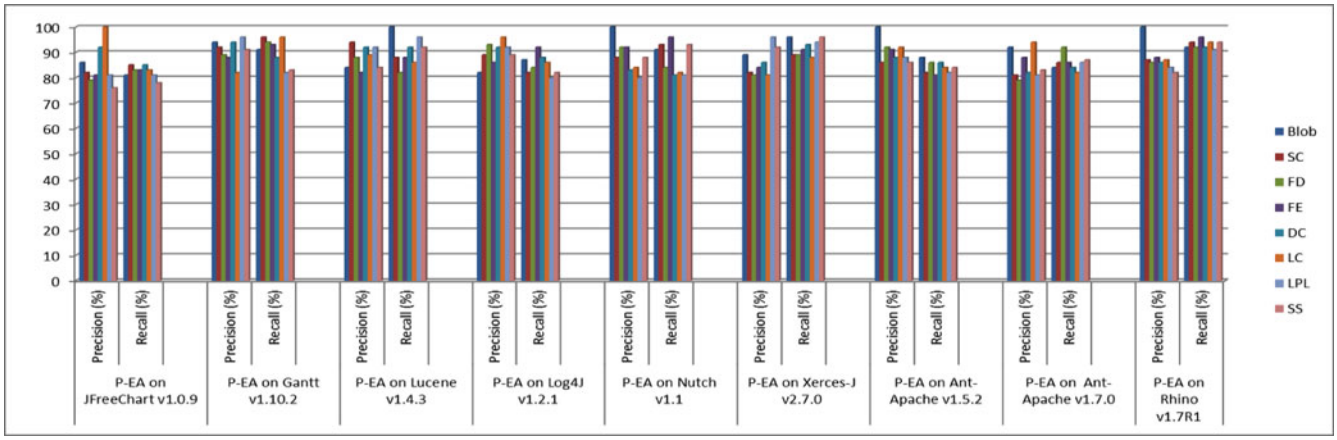


Fig. 9. Distribution of detected code-smells on the nine systems.

89 percent), i.e., most of the detected code-smells are correct. This confirms that our P-EA precision results are independent from the size of the systems to evaluate. For JFreeChart, the precision using P-EA is the lowest (84 percent) but still acceptable. JFreeChart contains a high number of functional decomposition and feature envy anti-patterns that are difficult to detect using metrics but with the deviance with reference code thus more code-smells can be detected if we use a lower severity score than 75 percent. For the same dataset, we can conclude that our P-EA approach performs much better (with a 95 percent confidence level) than existing single-population approaches (GP and GA) on almost all systems since the median precision scores are considerably higher. In fact, P-EA provides better results since some code-smells cannot be detected using only one fitness function based on metrics or deviance from reference code examples. P-EA can detect and rank the detected classes based on the use of both detectors and detection rules that maximizes the consensus. The RS did not perform well in terms of precision due to the huge search-spaces of detectors and rule combinations to explore.

According to Table 3, The average recall score of P-EA on the different systems is around 87 percent (better than precision). JFreeChart has the lowest recall score with 81 percent. Single-population approaches (GP and GA) provide also good results (an average of 74 percent) but lower than P-EA ones. Our approach is significantly different from existing ones that are rule-based and/or single-population-based. A key problem with existing single population approaches, which are based on detection rules, is that they simplify the different notions/symptoms that are useful for the detection of certain code-smells. In particular, to detect blobs, the notion of size is important. Most size metrics are highly correlated to each others, and the best measure of size can depend on the system itself. In this case, the use of different algorithms in parallel (not only metrics-based algorithms) can help to detect most of the code-smells more efficiently than a single population approach using quality metrics.

As interesting observation from the results of Tables 2 and 3 is that the medians are close, the results are statistically different but the effect size [56], [57] which quantifies the difference is small for most of the systems and techniques considered in our experiments. The Wilcoxon rank sum 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 [57]. The effect size is considered: (1) *small* if $0.2 \geq d < 0.5$, (2) *medium* if $0.5 \leq d < 0.8$, or (3) *large* if $d \leq 0.8$. In conclusion, our technique is able to accurately identify design code-smells (*RQ1*) more accurately than existing single-population approaches (*RQ3*). The statistical analysis of the obtained results using the Wilcoxon test confirms these experimental statements on almost all systems, especially for medium and large systems.

Based on the results of Fig. 9, we noticed that our technique does not have a bias towards the detection of specific code-smells types. As described in Fig. 9, in all systems, we had an almost equal distribution of each code-smell types. On some systems such as Nutch, the distribution is not as balanced. This is principally due to the number of actual code-smell types in the system. Having a relatively good distribution of code-smells is useful for a quality engineer. Overall, all the eight code smell types are detected with good precision and recall scores in the different systems (more than 80 percent). This ability to identify different types of code-smells 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 a 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. This difficulty limits the performance of GP in well detecting this type of code-smells. Thus, we can conclude that our P-EA approach detects well all the types of considered code-smells (RQ2).

Since it is not sufficient to compare our proposal with only search-based work, we compared the performance of P-EA with DECOR [16] and JDeodorant [49]. 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. JDeodorant is an Eclipse plug-in that identifies two main types of bad-smells using a set of quality metrics: blob and feature envy (FE). The results of the execution of DECOR and

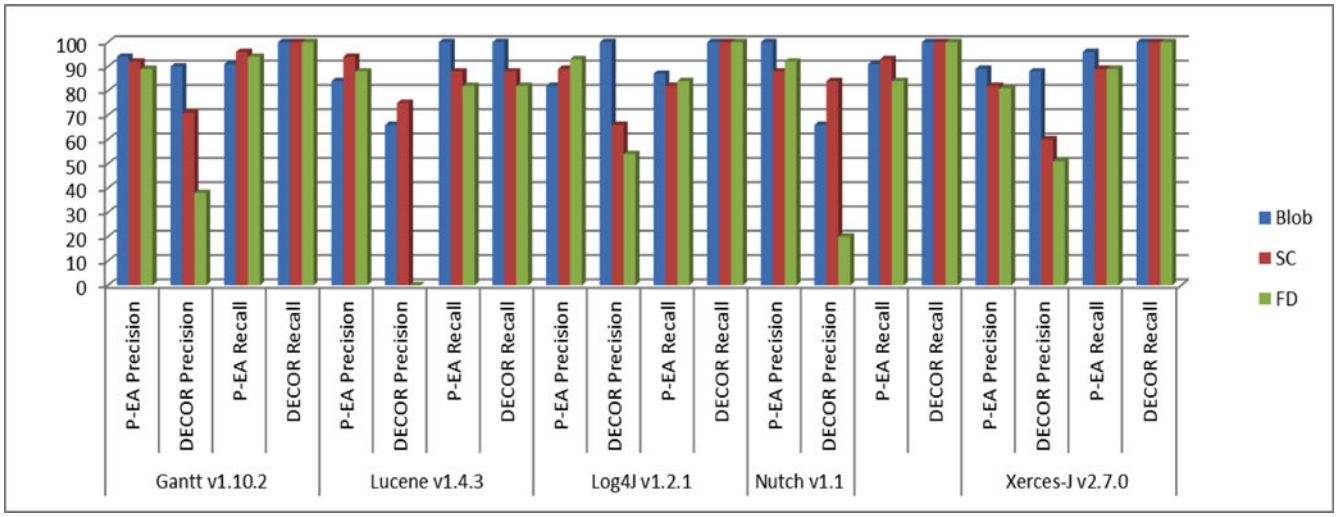


Fig. 10. The average precision and recall scores of P-EA and DECOR obtained on GanttProject, Nutch, Log4J, Lucene, and Xerces-J based on three code-smell types (Blob, SC, and FD).

JDeodorant are only available for the following systems: GanttProject, Nutch, Log4J, Lucene and Xerces-J. Figs. 10 and 11 summarize the results of the precision and recall obtained on the above-mentioned five systems. The recall for DECOR in all the systems is 100 percent signifying that it is better than P-EA, however the precision scores are lower than our proposal on all systems. For example, the average precision to detect functional decompositions using DECOR is lower than 25 percent, whereas we can detect this type of code-smells with more than 90 percent of precision. The same observation is valid for the spaghetti-code, BLOP is able to detect SC with more than 83 percent in terms of precision however DECOR detected the same type of code-smell with a precision lower than 62 percent. 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 P-EA is lower than DECOR, in average, but it is still acceptable, i.e., higher than 87 percent. The same observations are also valid for JDeodorant as described in Fig. 11. P-EA outperforms JDeodroant in terms of precision and recall on all the evaluated systems. The average precision and recall of JDeodorant for both code-smell types are respectively 76 and 73 percent. To conclude, our P-EA proposal also outperforms, in average, an existing approach not based on meta-heuristic search (RQ4).

6 Discussions

In this section, we discuss different issues concerning our P-EA detection approach. An important factor to our detection technique is the severity threshold value that to decide if a detected class can be considered as a code-smell or not. In Fig. 12, we present the precision and recall scores of our approach when varying the severity threshold (St) with $St \in \{95, 90, 85, 80, 75, 70, 65\}$. The figure shows that the precision of our approach improves as we consider a higher severity score whereas the recall score decreases and vice-versa. In fact, we observe that 75 percent of severity threshold represents a good trade-off between precision and recall scores. We can generalize this threshold since we evaluate the

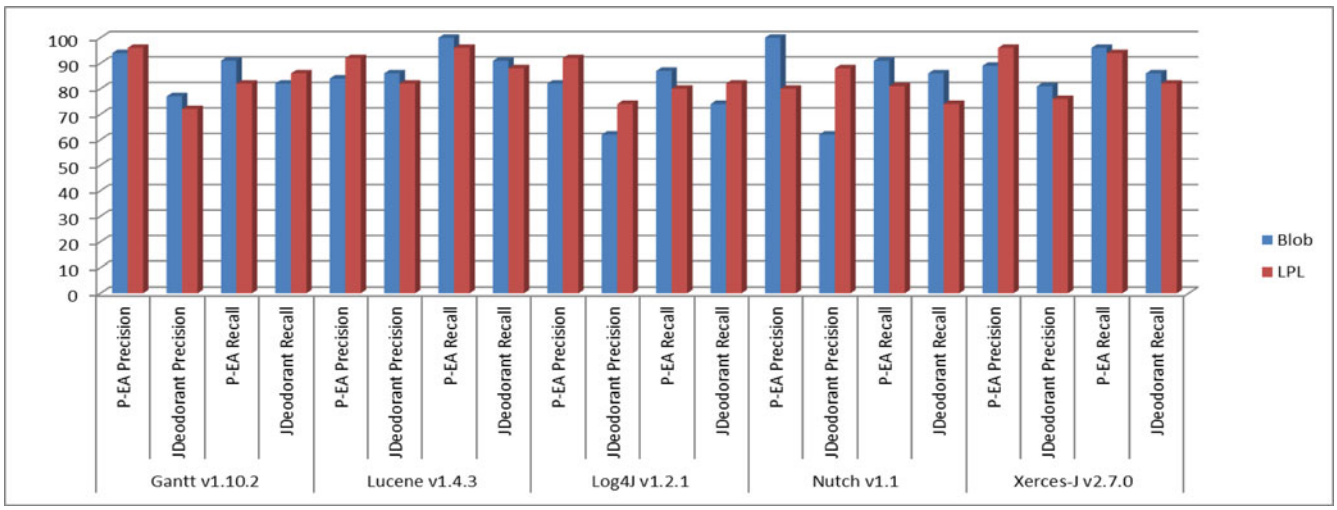


Fig. 11. The average precision and recall scores of P-EA and JDeodorant obtained on GanttProject, Nutch, Log4J, Lucene, and Xerces-J based on two code-smell types (Blob and LPL).

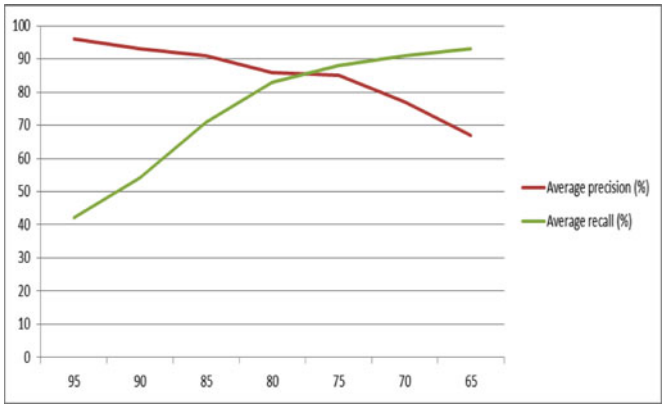


Fig. 12. The impact of the severity threshold on detection results (average precision and recall scores on all systems to evaluate).

different *St* values on the eight systems considered in the experiments. However, developers can choose other threshold values if they want to increase precision in some contexts (for example, to reduce the manual inspection time). Another observation is that our severity score represents a good indication of the risk of detected classes. In fact, all classes ranked in the top correspond to a real code-smell since the precision becomes lower if the severity threshold decreases.

Fig. 13 shows that the performance of our approach improves as we increase the percentage of best solutions (from each population of detectors and rules) for intersection at each iteration. However, the results become stable after 5 percent. For this reason, we considered this threshold in our experiments. Our P-EA technique requires the comparison of every selected GP solution to every selected GA solution, thus the execution time needs to be considered (number of comparison). Indeed, we believe that 5 percent are a good threshold value for a population of 100 individuals (around 25 comparisons per iteration) to keep reasonable execution time.

The reliability of the proposed approach requires an example set of good code and code-smell examples. It can be argued that constituting such a set might require more work than identifying and adapting code-smells detection rules. In our study, we showed that by using JHotdraw directly, without any adaptation, the P-EA method can be used out of the box and this will produce good detection results for the detection of code-smells for

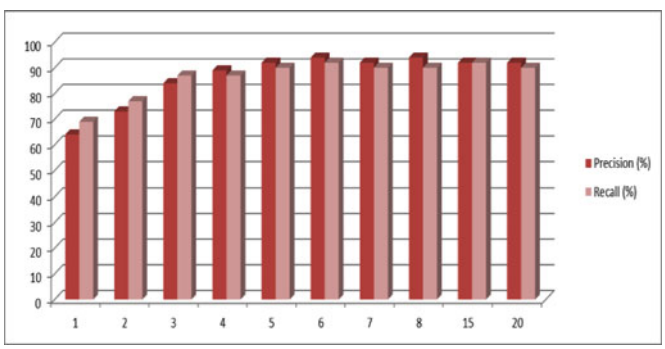


Fig. 13. The impact of percentage of best solutions selected for the intersection process on detection results (average precision and recall scores on all systems to evaluate).

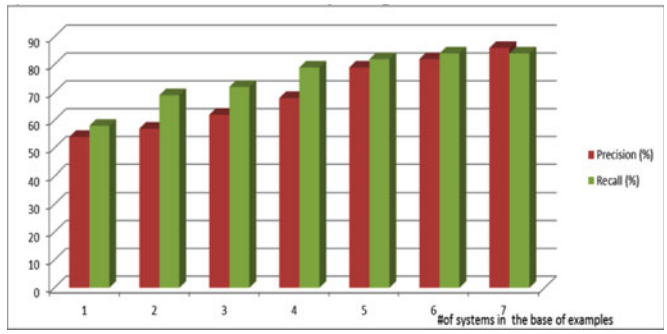


Fig. 14. The impact of the number of code-smell examples on detection results (precision and recall scores are calculated on GanttProject).

the eight 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. Fig. 14 shows that only code-smell examples extracted from five different open source systems can be used to obtain good precision and recall scores.

The detection results might vary depending on the search space exploration, since solutions are randomly generated, though guided by a meta-heuristic. To ensure that our results are relatively stable, we compared the results of multiple executions for P-EA on GanttProject as shown in Fig. 15 (over 51 runs). Based on the statistical analysis, described previously, similar observations were obtained with the other projects regarding the stability. We believe that our approach is stable, since the scores are approximately the same for 51 different executions on the different systems. In addition, the statistical analysis performed, during the comparison against the other algorithms, confirms the stability of our results.

Usually in the optimization research field, the most time consuming operation is the evaluation step [7]. Thus, we show how our P-EA is more efficient than the GA and GP from a CPU time viewpoint. In fact, all the algorithms under comparison were executed on machines with Intel Xeon 3 GHz processors and 4 GB RAM. We note that each of GA and GP were run on a single machine. However, our P-EA was executed on two nodes (machines) following the previously described parallel model. We recall that all algorithms were run for 100,000 evaluations. This allows us to make a

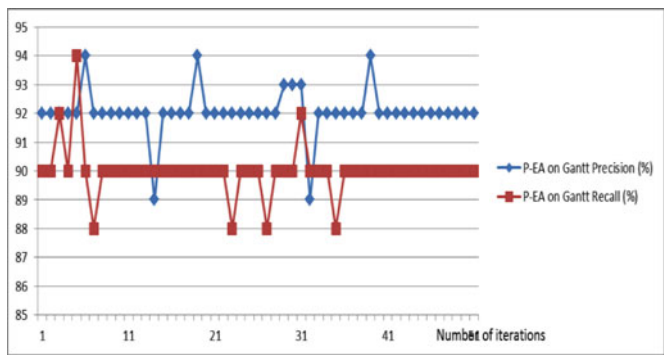


Fig. 15. Execution of P-EA algorithm over 51 runs on GanttProject.

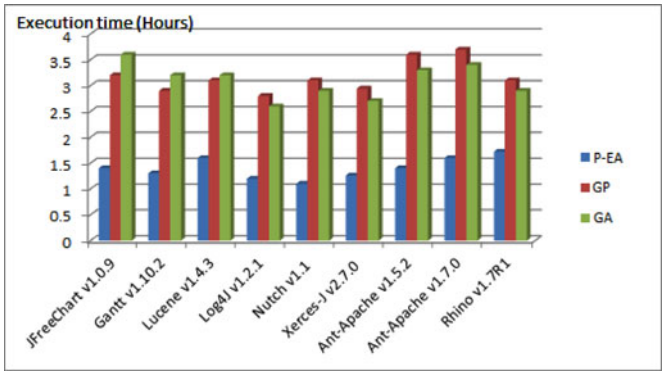


Fig. 16. Average execution time comparison (over 51 runs) on the different systems.

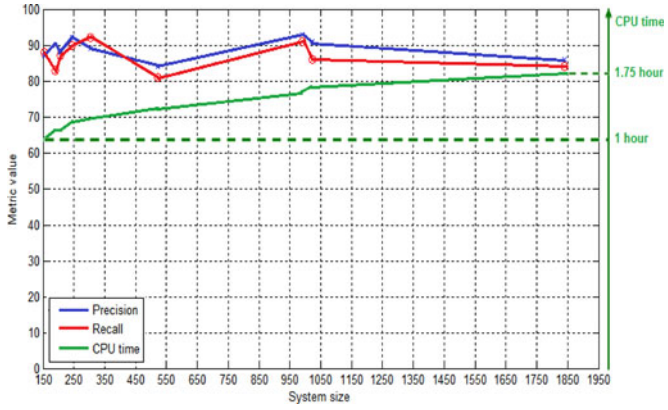


Fig. 17. Scalability of P-EA with respect to system size.

fair comparison between CPU times. Fig. 16 illustrates the obtained CPU times of all algorithms on each of the considered systems (cf. Table 3). We note that the results presented in this figure were analyzed by using the same previously described statistical analysis methodology. In fact, based on the obtained p-values regarding CPU times, the P-EA is demonstrated to be faster than GA and GP as highlighted through Fig. 16. The P-EA spends approximately the half amount of time required for GA or GP. This observation could be explained by the fact that the performed 100,000 evaluations are distributed between the two machines (50,000 for each). In this way, the P-EA is able to evaluate 200 individuals at each iteration, which is not the case for GA or GP which evaluates only 100 individuals at each iteration. We can see that parallelization seems to be an interesting approach to tackle software engineering problems where the individual evaluations are expensive like the code-smells detection problem.

Fig. 17 illustrates the evolution of precision, recall and CPU time with respect to the increase of system size (number of classes). We see from this figure that the precision and recall values are stable even if the system size increases. The same observation could be seen for CPU time which is between 1 hour and 1 hour and 45 minutes. We can say that PEA is scalable with respect to system size since it gives high precision and recall values for an acceptable execution time.

One of the main reasons explaining the outperformance of our technique relative to DECOR and JDeodroant is that they use relatively permissive constraints/rules, which sacrifice the precision and guarantee a high recall. In addition, these tools do not provide a ranking of the severity of the detected code smells. Thus, the classes that do not satisfy the rules are not considered as code smells (deterministic process). However, in our technique, we rank the detected classes based on the severity score, and this can explain the high-precision scores obtained by our technique. Furthermore, some types of code smells such as the functional decomposition are difficult to detect using quality metrics. The use of two different detection techniques in parallel explains the detection of code smells that are hard to detect using only quality metrics (GP). Another observation is related to the systems used in our experiments where some of them have different programming contexts. This explains why the performance of P-EA is different on the systems under evaluation since the results depend on the base of examples that should contain a diversified set of systems with different programming contexts to generate good quality of rules and detectors. However, this is an issue that can be fixed by improving the quality of the base of examples and classifying the systems based on their size, context, etc. The base of examples is updated automatically with new code smell examples detected by our tool on new systems after validating them manually by software engineers. Thus, after several revisions of the base of examples, the P-EA algorithm can be executed to refine the rules and detectors. A software company can use our tool using their previous projects as a training set to generate a high quality of rules and detectors that take into consideration the programming behaviour of their developers. However, our experiments show that open source systems can be considered as a good starting point.

7 Threats to Validity

Following the methodology proposed by Wohlin et al. [53], there are four types of threats 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 rank sum test with a 95 percent confidence level to test if significant differences exist 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 our comparison with the techniques not based on heuristic search, we considered the parameters provided with the tools. This is can be considered as a threat that can be addressed in the future by evaluating the impact of different parameters on the quality of the results of DECOR and JDeodorant.

Internal validity is concerned with the causal relationship between the treatment and the outcome. We consider the internal threats to validity in the use of stochastic algorithms since our experimental study is performed based on 51 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test [35] with a 95 percent

confidence level ($\alpha = 5\%$). However, 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. The parameters' values used in our experiments are found by trial-and-error method, which is commonly used in the SBSE community [55]. However, it would be an interesting perspective to design an adaptive parameter tuning strategy [54] for our approach so that parameters are updated during the execution in order to provide the best possible performance.

Construct validity is concerned with the relationship between theory and what is observed. Most of what we measure in our experiments are standard metrics such as precision and recall that are widely accepted as good proxies for quality of code-smells detection solutions. The notion of "intersection" we use in this paper is new and so constitutes a possible threat to construct validity. However, we discussed in the experimentations section several threshold values regarding the number of solutions considered at each iteration for intersection. We also used two evolutionary algorithms in our P-EA approach. We do not anticipate that very different results would be obtained using other meta-heuristics algorithm with the same problem formulation. In our future work, we plan to compare the performance of P-EA using different meta-heuristics search algorithms. Another construct validity threat is related to the absence of similar work that uses distributed evolutionary algorithms for code-smells detection. For that reason, we compare our proposal with other existing techniques not based on distributed algorithms. Another threat to construct validity arises because, although we considered eight types of code-smells, we did not evaluate the detection of other types of code-smells. In future work, we plan to evaluate the performance of our P-EA proposal to detect some other types of code-smell. Another construct threat can 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.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on eight different widely used open-source systems belonging to different domains and with different sizes, as described in Table 1. However, we cannot assert that our results can be generalized to industrial Java applications, other programming languages, and to other practitioners. Future replications of this study are necessary to confirm our findings.

8 Related Work

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. Nevertheless, there is no work that focuses on combining different detection algorithms to find a consensus when identifying code-smells.

In this paper, we classify existing approaches for code-smells detection into seven broad categories: manual approaches, symptom-based approaches, rule-based approaches, probabilistic approaches, visualization-based approaches, search-based approaches and cooperative-based approaches.

8.1 Manual Approaches

In the literature, the first book that has been specially written for design smells was by Brown et al. [13] which provide broad-spectrum and large views on design smells, and anti-patterns that aimed at a wide audience for academic community as well as in industry. Indeed, in [25], Fowler et al. have described a list of design smells which may exist in a program. They suggested that 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. [36] have also proposed a manual approach for detecting code-smells in object-oriented designs. The idea is to create a set of "reading techniques" which 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. So that, each reading technique helps the maintainer focusing on some aspects of the design, in such a way that an inspection team applying the entire family should achieve a high degree of coverage of the design code-smells. In addition, in [37], 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. However, the majority of the detected problems were simple ones, since it is based on simple conditions with particular threshold values. As a consequence, this approach did not address complex design code-smells.

The main disadvantage of exiting manual 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. Another important issue is that locating code-smells manually has been described as more a human intuition than an exact science. To circumvent the above-mentioned problems, some semi-automated approaches have emerged.

8.2 Symptom-Based Detection

Moha et al. [16], [17] started by describing code-smell symptoms using a domain-specific-language 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. 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. Similarly, Munro [38] have proposed description and

symptoms-based approach using a precise definition of bad smells from the informal descriptions given by the originators Fowler et al. [25]. The characteristics of design code-smells have been used to systematically define a set of measurements and interpretation rules for a subset of design code-smells as a template form. This template consists of three main parts: a code smell name, a text-based description of its characteristics, and heuristics for its detection.

The major limitation of symptoms-based approaches is that there exists no consensus in defining symptoms. A code-smell may have several and different interpretations by a maintainer. Another limitation is that for an exhaustive list of code-smells, the number of possible code-smells to be manually described, characterized with rules and mapped to detection algorithms can be very large. Indeed, the background and knowledge of maintainers affect their understanding of code-smells, given a set of symptoms. As a consequence, symptoms-based approaches are also considered as time-consuming and error-prone. Thus automating the detection of design code-smells is still a real challenge.

8.3 Metric-Based Approaches

The idea to automate the problem of design code-smells detection is not new; neither is the idea to use quality metrics to improve the quality of software systems.

Marinescu [19] 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, Munro [38] proposed metric-based heuristics for detecting code-smells, which are similar to Marinescu's detection strategies. Munro has also performed an empirical study to justify his choice of metrics and thresholds for detecting smells. Salehie et al. [40] proposed a metric-based heuristic framework to detect and locate object-oriented design flaws similar to those illustrated by Marinescu [14]. 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 and Lewerentz [41] 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.

In general, the effectiveness of combining metric/threshold is not obvious. That is, for each code-smell, rules that are expressed in terms of metric combinations need a significant calibration effort to find the fitting threshold values for each metric. Since there exists no consensus in defining design smells, different threshold values should be tested to find the best ones.

8.4 Probabilistic Approaches

Probabilistic approaches represent another way for detecting code-smells. Alikacem and Sahraoui [42] have considered the code-smells detection process as a fuzzy-logic problem, using rules with fuzzy labels for metrics, e.g., small, medium, large. To this end, they proposed a domain-specific language that allows the specification of fuzzy-logic rules that include quantitative properties and relationships among classes. The thresholds for quantitative properties are replaced by fuzzy labels. Hence, when evaluating the rules, actual metric values are mapped to truth values for the labels by means of membership functions that are obtained by fuzzy clustering. Although, fuzzy inference allows to explicitly handle the uncertainty of the detection process and ranks the candidates, authors did not validate their approach on real programs. Recently, another probabilistic approach has been proposed by Khomh et al. [20] extending the DECOR approach [16], a symptom-based approach, to support uncertainty and to sort the code-smell candidates accordingly. This approach is managed by Bayesian belief network (BBN) that implements the detection rules of DECOR. 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.

8.5 Visualization-Based Approaches

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. [45] present a pattern-based framework for developing tool support to detect software anomalies by representing potential code-smells with different colors. Dhambri et al. [46] have proposed a visualization-based approach to detect design anomalies by automatically detecting some symptoms and letting others to 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 [39] are based on manual and human inspection, they still, not only, slow and time-consuming, but also subjective.

8.6 Search-Based Approaches

Our approach is inspired by contributions in the domain of Search-Based Software Engineering [24], [47]. SBSE uses search-based approaches to solve optimization problems in software engineering. Once a software engineering task is framed as a search problem, many search algorithms can be applied to solve that problem. In [42], we have proposed

another approach, based on search-based techniques, for the automatic detection of potential code-smells in code. The detection is based on the notion that the more code deviates from good practices, the more likely it is bad. In another work [21], 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. In this work, we combine between our two previous work [21], [22] using P-EA to detect code-smells.

Machine learning represents another alternative for detecting design code-smells. Catal et al. [51] used different machine learning algorithms to predict defective modules. They investigated the effect of dataset size, metrics set, and feature selection techniques for software fault prediction problem. They employed several algorithms based on artificial immune systems (AIS). Kessentini et al. [22] have proposed a code-smell detection algorithm based on the idea that the more code deviates from good practices, the more likely it is bad. This approach learns from examples of well-designed and implemented software elements, to estimate the risks of classes to deviate from "normality", i.e., a set of classes representing "good" design that conforms to object-oriented principles. Elements of assessed systems that diverge from normality to detectors are considered as risky. Although this approach succeeded in discovering risky code, it does not provide a mechanism to identify the type of the detected code-smell. Similarly, Ren et al. [52] have proposed an approach for detecting design smells using machine learning technique inspired from the AIS. Their approach is designed to systematically detect classes whose characteristics violate some established design rules. Rules are inferred from sets of manually-validated examples of code-smells reported in the literature and freely-available. The major benefit of machine-learning based approaches is that it does not require great experts' knowledge and interpretation. In addition, they succeeded, to some extent, to detect and discover potential code-smells by reporting classes that are similar (even not identical) to the detected code-smells. However, these approaches depend on the quality and the efficiency of data, i.e., code-smell instances, to learn from. Indeed, the high level of false positives represents the main obstacle for these approaches.

Based on recent SBSE surveys [47], the use of parallel metaheuristic search is still very limited in software engineering. Indeed, there is no work that uses cooperative parallel metaheuristic search to detect code smells. This is the first adaptation of cooperative parallel metaheuristics to solve a software engineering problem.

8.7 Cooperative-Based Approaches in Software Engineering

Some cooperative approaches to address software engineering problems have been proposed recently. Arcuri and Yao [50] handled the bug fixing problem using a competitive co-evolutionary approach in which programs and test cases co-evolve, influencing each other with the aim of fixing the maximum number of bugs in the programs. Adamopoulos et al. [51] tackled the mutation testing problem using a competitive co-evolutionary approach. The goal is to improve the efficiency of generated test cases by evaluating their capabilities to kill mutants (introduced errors in the system). Ren et al. [52] proposed a cooperative co-evolutionary approach for software project staff assignments and job scheduling to minimize the completion time. The two main tasks that are handled in parallel are finding the best sequence of work packages and staff assignments to teams.

Our P-EA proposal is different from existing co-evolutionary approaches. First, we propose this is the first time that the problem of code-smells detection is formulated in a cooperative way. Second, existing co-evolutionary approaches are composed by two populations addressing two different problems. However, our P-EA proposal is based on two populations that are addressing the same problem from different perspectives. Third, the P-EA formulation presented in this paper is based on the notion of "intersection" which is not used by existing work. Finally, the two populations are executed on two different machines in parallel in our P-EA framework. However, most of the co-evolutionary approaches are implemented in a sequential way (co-evolution).

9 Conclusion

In this paper, we proposed a new search-based approach for code-smells detection. In our cooperative parallel metaheuristic adaptation, two populations evolve simultaneously with the objective of each depending upon the current population of the other in a cooperative manner. The first population generates a set of detection rules using genetic programming and simultaneously a second population tries to detect code-smells using a deviation from examples of well-designed codes. Both populations are executed, on the same system to evaluate, and the solutions are penalized based on the consensus found, i.e.: intersection between the detection results of both populations. The best detection results will be the code-smells detected by the majority of the algorithms. The statistical analysis of the obtained results provides compelling evidence that cooperative P-EA outperforms single population evolution and random search based on a benchmark of several large open-source systems.

Future work should validate our approach with additional code-smell types in order to conclude about the general applicability of our methodology. Furthermore, in this paper, we only focused on the detection of code-smells. We are planning to extend the approach by automating the correction of code-smells. In addition, we will consider the importance of code-smells during the detection step using previous code-changes, classes-complexity, etc. Thus, the detected code-smells will be ranked not only based on the severity score but also an importance score. We will evaluate also the use of more than two algorithms executed in parallel as part of our future work. An increase of the number of algorithms executed in parallel can have a negative impact on the quality of the results since it will be more difficult to find a consensus between them. Another future research direction related to our work is to adapt our parallel evolutionary approach to several other software

engineering problems such as software testing and the next release problem.

REFERENCES

[1] P. Siarry and Z. Michalewicz, *Advances in Metaheuristics for Hard Optimization (Natural Computing Series)*. New York, NY, USA: Springer, 2008.

[2] E. Alba, *Parallel Metaheuristics: A New Class of Algorithms*. Hoboken, NJ, USA: Wiley, 2005.

[3] T. Burczyńskia, W. Kuśa, A. Długosza, and P. Oranteka, "Optimization and defect identification using distributed evolutionary algorithms," *Eng. Appl. Artif. Intell.*, vol. 4, no. 17, pp. 337–344, 2004.

[4] X. Li and X. Yao, "Cooperatively coevolving particle swarms for large scale optimization," *IEEE Trans. Evol. Comput.*, vol. 16, no. 2, pp. 210–224, Apr. 2012.

[5] E. J. Hughes, "Evolutionary many-objective optimization: Many once or one many?," in *Proc. IEEE Congr. Evol. Comput.*, 2005, pp. 222–227.

[6] Y. Wang, Z. Cai, G. Guo, and Y. Zhou, "Multiobjective optimization and hybrid evolutionary algorithm to solve constrained optimization problems," *IEEE Trans. Syst., Man, Cybern., Part B: Cybern.*, vol. 37, no. 3, pp. 560–575, Jun. 2007.

[7] E-G. Talbi, *Metaheuristics: From Design to Implementation*. Hoboken, NJ, USA: Wiley, 2009.

[8] C. Salto and E. Alba, "Designing heterogeneous distributed GAs by efficiently self-adapting the migration period," *Appl. Intell.*, vol. 36, no. 4, pp. 800–808, 2012.

[9] M. Tomassini and L. Vanneschi, "Guest editorial: Special issue on parallel and distributed evolutionary algorithms, part two," *Genetic Program. Evolvable Mach.*, vol. 11, no. 2, pp. 129–130, 2010.

[10] M. Kessentini, H. A. Sahraoui, M. Boukadoum, and O. Ben Omar, "Search-based model transformation by example," *Softw. Syst. Model.*, vol. 11, no. 2, pp. 209–226, 2012.

[11] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA,USA: Addison Wesley, 1989.

[12] A. Abran and H. Hguyenkim, "Measurement of the maintenance process from a demand-based perspective," *J. Softw. Maintenance: Res. Practice*, vol. 5, no. 2, pp. 63–90, 1993.

[13] W. J. Brown, R. C. Malveau, W. H. Brown, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. Hoboken, NJ, USA: Wiley, 1998.

[14] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proc. 20th Int. Conf. Softw. Maintenance.*, 2004, pp. 350–359.

[15] N. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. Int. Thomson Comput. Press, London, UK, 1997.

[16] N. Moha, Y. G. Guéhéneuc, L. Duchien, and A. F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan./Feb. 2010.

[17] N. Moha and Y. G. Guéhéneuc, "Decor: A tool for the detection of design defects," in *Proc. Int. Conf. Autom. Softw. Eng.*, 2007, pp. 527–528.

[18] H. Liu, L. Yang, Z. Niu, Z. Ma, and W. Shao, "Facilitating software refactoring with appropriate resolution order of bad smells," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 265–268.

[19] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proc. Int. Conf. Softw. Maintenance*, 2004, pp. 350–359.

[20] F. Khomh, S. Vaucher, Y. G. Guéhéneuc, and H. A. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Proc. Int. Conf. Quality Softw.*, 2009, 305–314.

[21] M. Kessentini, W. Kessentini, H. A. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *Proc. IEEE 19th Int. Conf. Program Comprehension*, 2011, pp. 81–90.

[22] M. Kessentini, S. Vaucher, and H. A. Sahraoui, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code," in *Proc. Int. Conf. Autom. Softw. Eng.*, 2010, pp. 113–122.

[23] A. Ouni, M. Kessentini, H. A. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: A multi-objective approach," *Autom. Softw. Eng.*, vol. 20, no. 1, pp. 47–79, 2012.

[24] M. Harman, "The current state and future of search based software engineering," in *Proc. Future Softw. Eng.*, 2007, pp. 342–357.

[25] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Reading, MA,USA: Addison Wesley, 1999.

[26] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 293–318, Jun. 1994.

[27] W. Banzhaf, "Genotype-phenotype-mapping and neutral variation: A case study in genetic programming," in *Proc. Int. Conf. Parallel Problem Solving from Nature*, 1994, pp. 322–332.

[28] L. Nanni and A. Lumini, "Generalized Needleman-Wunsch algorithm for the recognition of t-cell epitopes," *Expert Syst. Appl.*, vol. 35, no. 3, pp. 1463–1467, 2008.

[29] M. Mitchell, An Introduction to Genetic Algorithms. Cambridge, MA, USA: MIT Press, 1998.

[30] H. Hou and G. Dozier, "An evaluation of negative selection algorithm with constraint-based detectors," in *Proc. 44th Annu. ACM Southeast Regional Conf.*, 2006, pp. 134–139.

[31] M. Brudno, "*Algorithms for comparison of DNA sequences*," Ph.D. dissertation, Comput. Sci. Dept., Stanford Univ., Stanford, CA, USA, 2004.

[32] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.

[33] M. Kim, V. Sazawal, and D. Notkin, "An empirical study of code clone genealogies," in *Proc. Joint Meeting Eur. Soft. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2005, pp. 187–196.

[34] (2014) [Online]. Available: http://www.jhotdraw.org/

[35] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *J. Amer. Statist. Assoc.*, vol. 47, no. 260, pp. 583–621, 1952.

[36] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: Using reading techniques to increase software quality," in *Proc. Int. conf. Object-Oriented Program., Syst., Languages, Appl.*, 1999, pp. 47–56.

[37] O. Ciupke, "Automatic detection of design problems in object-oriented reengineering," in *Proc. Int. Conf. Technol. Object-Oriented Language Syst.*, 1999, pp. 18–32.

[38] M. J. Munro, "Product metrics for automatic identification of 'Bad Smell' design problems in Java source-code," in *Proc. IEEE 11th Int. Softw. Metrics Symp.*, 2005, pp. 15–15.

[39] G. Langelier, H. A. Sahraoui, P. Poulin, "Visualization-based analysis of quality for large-scale software systems," in *Proc. Int. Conf. Autom. Softw. Eng.*, 2005, pp. 214–223.

[40] M. Salehie, S. Li, and L. Tahvildari, "A metric-based heuristic framework to detect object-oriented design flaws," in *Proc. IEEE 14th Int. Conf. Program Comprehension*, 2006, pp. 159–168.

[41] K. Erni and C. Lewerentz, "Applying design metrics to object-oriented frameworks," in *Proc. IEEE 3rd Int. Softw. Metrics*, 1996, pp. 64–74.

[42] H. Alikacem and H. A. Sahraoui,"Détection d'anomalies utilisant un langage de description de règle de qualité," in *Proc. Int. Conf. Languages Models Objects*, 2006, pp. 185–200.

[43] C. Catal and B. Diri, "Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem," *Inform. Sci.*, vol. 179, no. 8, pp. 1040–1058, 2009.

[44] S. Hassaine, F. Khomh, Y. G. Guéhéneuc, and S. Hamel, "IDS: An immune-inspired approach for the detection of software design smells," in *Proc. Int. Conf. Quality Inf. Commun. Technol.*, 2010, pp. 343–348.

[45] S. C. Kothari, L. Bishop, J. Sauceda, and G. Daugherty, "A pattern-based framework for software anomaly detection," *Softw. Quality J.*, vol. 12, no. 2, pp. 99–120, 2004.

[46] K. Dhambri, H. A. Sahraoui, and P. Poulin, "Visual detection of design anomalies," in *Proc. Int. Softw. Maintenance Reeng.*, 2008, pp. 279–283.

[47] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, 61 pages.

[48] I. G. Czibula and G. Czibula, "Clustering based automatic refactorings identification," in *Proc. Int. Symp. Symbolic Numeric Algorithms Sci. Comput.*, 2008, pp. 253–256.

[49] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of type-checking bad smells," in *Proc. Int. Conf. Softw. Maintenance Reeng.*, 2008, pp. 329–331.

[50] A. Arcuri, X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Proc. IEEE Congr. Evol. Comput.*, 2008, pp. 162–168.

[51] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Proc. Genetic Evol. Comput. Conf.*, 2004, 1338–1349.
[52] J. Ren, M. Harman, M. Di Penta, "Cooperative co-evolutionary optimization of Software project staff assignments and job scheduling," in *Proc. Int. Symp. Search-Based Softw. Eng.*, 2011, pp. 127–141.
[53] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Norwell, MA, USA: Kluwer, 2000.
[54] G. Karafotias, M. Hoogendoorn, and A. E. Eiben, "Why parameter control mechanisms should be benchmarked against random variation," in *Proc. IEEE Congr. Evol. Comput.*, 2013, pp. 349–355.
[55] A. E. Eiben and S. K. Smit, "Parameter tuning for configuring and analyzing evolutionary algorithms," *Swarm Evol. Comput.*, vol. 1, no. 1, pp. 19–31, 2011.
[56] F. Ferrucci, M. Harman, J. Ren, F. Sarro, "Not going to take this anymore: Multi-objective overtime planning for software engineering projects," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 462–471.
[57] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Evanston, IL, USA: Routledge, 1988.

Wael Kessentini received the BSc degree from the University of Tunis in 2011 and the MSc degree in 2013. Currently, he is working toward the PhD degree at the University of Montreal. He was a research assistant in the Search-Based Software Engineering at Michigan in 2013. His main research interests include search-based software engineering, software quality, refactoring, and model-driven engineering. He is a student member of the IEEE.

Marouane Kessentini received the PhD degree in computer science from the University of Montreal, Canada, in 2011. He is a tenure-track assistant professor at the University of Michigan. He is the founder of the research group: Search-Based Software Engineering (SBSE) at Michigan. His research interests include the application of artificial intelligence techniques to software engineering (SBSE), software testing, model-driven engineering, software quality, and reengineering. He has published around 50 papers in conferences, workshops, books, and journals including three Best Paper Awards. He was a program-committee/organization member in several conferences and journals. He was the cochair of the SBSE track at GECCO2014, and the founder of the North American Search-Based Software Engineering Symposium (NasBASE). He is a member of the IEEE.

Houari A. Sahraoui received the diploma in engineering from the National Institute of Computer Science, Algiers, in 1990, and the PhD degree in computer science, Pierre & Marie Curie University LIP6, Paris, in 1995. He is full a professor at the Department of Computer Science and Operations Research (GEODES, Software Engineering Group) of the University of Montreal. Before joining the university, he held the position of a lead researcher of the Software Engineering Group at CRIM (Research Center on Computer Science, Montreal). His research interests include the application of artificial intelligence techniques to software engineering, object-oriented metrics and quality, software visualization, and reengineering. He has published around 100 papers in conferences, workshops, books, and journals, edited three books, and gives regularly invited talks. He was a program committee member in several major conferences, a member of the editorial boards of two journals, and an organization member of many conferences and workshops. He was the general chair of the IEEE Automated Software Engineering Conference in 2003. He is a member of the IEEE.

Sim Bechikh received the BSc degree in computer science applied to management, the MSc degree in modeling, and the PhD degree in computer science applied to management from the University of Tunis, Tunisia, in 2006, 2008, and 2013, respectively. He worked, for four years, as an attached researcher within the Optimization Strategies and Intelligent Computing Lab (SOIE), Tunisia. Currently, he is a postdoctoral researcher at the Search-Based Software Engineering (SBSE) at MST lab, Missouri. His research interests include multi-criteria decision making, evolutionary computation, multi-agent systems, portfolio optimization, and search-based software engineering. Since 2008, he published several papers in well-ranked journals and conferences. Moreover, he received the Best Paper Award of the ACM Symposium on Applied Computing 2010 in Switzerland among more than three hundred participants. Since 2010, he is a reviewer for several conferences such as ACM SAC and GECCO, and various journals such as *Soft Computing* and the *International Journal of Information Technology and Decision Making*. He is a member of the IEEE.

Ali Ouni received the BSc degree and MSc degree diploma in computer science from the Higher Institute of Applied Sciences and Technology (ISSAT), University of Sousse, Tunisia, in 2008 and 2010, respectively. He is currently working toward the PhD degree in computer science under the supervision of Pr. Marouane Kessentini (University of Michigan) and Pr. Houari Sahraoui (University of Montreal). He is a member of the Search-Based Software Engineering (SBSE) research laboratory, University of Michigan, and a member of the GEODES software engineering laboratory, University of Montreal, Canada. His research interest includes the application of artificial intelligence techniques to software engineering (search-based software engineering). Since 2011, he published several papers in well-ranked journals and conferences. He is a program committee member in several conferences and journals such as GECCO'14, CMSEBA'14, and NasBASE'15. He is a student member of the IEEE.