Competitive Coevolutionary Code-Smells Detection

Mohamed Boussaa[1], Wael Kessentini[1], Marouane Kessentini[1], Slim Bechikh[1,2], and Soukeina Ben Chikha[2]

[1] CS, Missouri University of Science and Technology Missouri, USA
{bm217,marouanek,wa235,bechikhs}@mst.edu
[2] University of Tunis Tunis, Tunisia
soukeina.benchikha@insat.rnu.tn

Abstract. Software bad-smells, also called design anomalies, refer to design situations that may adversely affect the maintenance of software. 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. Although these bad practices are sometimes unavoidable, they should be in general fixed by the development teams and removed from their code base as early as possible. In this paper, we propose, for the first time, the use of competitive coevolutionary search to the code-smells detection problem. We believe that such approach to code-smells detection is attractive because it allows combining the generation of code-smell examples with the production of detection rules based on quality metrics. The main idea is to evolve two populations simutaneously where the first one generates a set of detection rules (combination of quality metrics) that maximizes the coverage of a base of code-smell examples and the second one maximizes the number of generated "artificial" code-smells that are not covered by solutions (detection rules) of the first population. The statistical analysis of the obtained results shows that our proposed approach is promising when compared to two single population-based metaheuristics on a variety of benchmarks.

1 Introduction

In general, object oriented software systems need to follow some traditional set of design principles such as data abstraction, encapsulation, and modularity [8]. However, some of these non-functional requirements can be violated by developers for many reasons like inexperience with object-oriented design principles, deadline stress, and much focus on only implementing main functionality.

As a consequence, there has been much research focusing on the study of bad design practices, also called code-smells, defects, anti-patterns or anomalies [8, 9, 11] 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 [9]. 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 [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 [9]. When consensus exists, the same symptom could be associated to many code-smells types, which may compromise the precise identification of code-smell types. These difficulties explain a large portion of the high *false-positive* rates reported in existing research.

In this paper, we start from the observation that most of existing works related to the use of SBSE or machine learning techniques [12, 13] require a high number of code-smell examples (data) to provide efficient solutions that can be based on detection rules or classification algorithms. However, code-smells are not usually documented by developers (unlike bugs report). To this end, we introduce an alternative approach based on the use of a Competitive Co-Evolutionary Algorithm (*CCEA*) [2]. We believe that a *CCEA* approach to code-smells detection is attractive because it allows us to combine the generation of code-smell examples with the generation of detection rules based on quality metrics. We show how this combination can be formulated as two populations in a Competitive Co-evolutionary search. In *CCEA*, two populations of solutions evolve simultaneously with the fitness of each depending upon the current population of the other. The first population generates a set of detection rules (combination of quality metrics) that maximizes the coverage of a base of code-smell examples and simultaneously a second population tries to maximize the number of generated "artificial" code-smells that are not covered by solutions (detection rules) of the first population. The artificial code-smell examples are generated based on the notion of deviance from well-designed code fragments.

We implemented our *CCEA* approach and evaluated it on four systems [14, 15, 16, 17] using an existing benchmark [19, 13]. We report the results on the effectiveness and efficiency of our approach, compared to different existing single population-based approaches [12, 18]. The statistical analysis of our results indicates that the *CCEA* approach has great promise; *CCEA* significantly outperforms both random and single population-based approaches with an average of more than 80% of precision and recall based on an existing benchmark containing four large open source systems [14, 15, 16, 17].

The primary contributions of this paper can be summarized as follows: (1) the paper introduces a novel formulation of the code-smell's problem using Competitive Co-evolution and, to the best of our knowledge, this is the first paper in the literature to use competitive co-evolution to detect code-smells; (2) The paper reports the results of an empirical study with an implementation of our co-evolutionary approach, compared to existing single population approaches [12, 18]. The obtained results provide evidence to support the claim that competitive co-evolution is more efficient and effective than single population evolution.

The remainder of this paper is as follows: Section 2 presents the relevant background and the motivation for the presented work; Section 3 describes the search algorithm; an evaluation of the algorithm is explained and its results are discussed in Section 4; Section 5 is dedicated to related work. Finally, concluding remarks and future work are provided in Section 6.

2 Code-Smells Detection Overview

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 Definitions

Code-smells, also called design anomalies or design defects, refer to design situations that adversely affect the software maintenance. As stated by [9], 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 [20] and suggesting improvement solutions. In [20], Beck defines 22 sets of symptoms of code smells. These include large classes, feature envy, long parameter lists, and lazy classes. Each code-smell type is accompanied by refactoring suggestions to remove it. Brown et al. [9] define another category of code-smells that are documented in the literature, and named anti-patterns. In our approach, we focus on the three 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: It is a code with a complex and tangled control structure; Functional Decomposition: 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. 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 [19, 13].

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 [21]. This follows a long tradition of using software metrics to evaluate the quality of the design including the detection of code-smells [11]. The most widely-used metrics are the ones defined by [21]. These metrics include Depth of Inheritance Tree, Weighted Methods per Class, Cohesion and Coupling Between Objects (CBO), etc. 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, number of lines of code in a method, number of attributes in a class, number of methods, lack of cohesion in methods, number of accessors, and number of private fields. We are using in this paper these metrics to generate code-smell examples and also detection rules.

2.2 Detection Issues

Overall, there is no general consensus on how to decide if a particular design fragment is a code-smell. In fact, 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.

Most of existing work related to the use of SBSE or machine learning techniques require a high number of code-smell examples to provide efficient solutions that can be based on detection rules or classification algorithms. However, code-smells are not usually documented by developers (not like bugs for example that are documented in bug reports). Thus, it is difficult to find these code-smell examples except in few open-source systems that are evaluated manually.

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. Thus, it is important to identify the type of code-smells when detecting them to help developers to prioritize the list of detected code-smells.

3 Competitive Coevolution for Code-Smells Detection

This section address the different issues described in Section 2 using *CCEA*. We first present an overview of the competitive coevolution algorithms and, subsequently, provide the details of our adaptation of *CCEA* to detect code-smells.

3.1 Approach Overview

3.1.1 Competitive Co-evolution Algorithms

The idea of *Co-evolutionary algorithms* (*CCEAs*) comes from the biological observation which shows that co-evolving some number of species defined as collections of phenotypically similar individuals is more realistic than simply evolving a population containing representatives of one species. Hence, instead of evolving a population (globally or spatially distributed) of similar individuals representing a global solution, it is more appropriate to co-evolve subpopulations of individuals representing specific parts of the global solution [1]. There are two types of co-evolution in the related literature: (1) *Cooperation* and (2) *Competition*.

Cooperation consists in subdividing the problem at hand into different sub-problems of smaller sizes than the original one and then solving it in a cooperative manner. In fact, each sub-population helps the others with the aim to solve the original problem. Competition consists in making solutions belonging to different species competing with each others with the goal to create *fitter* individuals in each species. Since we are interested in this paper in competitive co-evolution, we just detail, in what follows, *competitive CCEAs*. Differently to canonic EAs, in competitive CCEAs the population is subdivided into a pre-specified number of sub-populations (each denoting a species) where the fitness value of a particular individual depends on the fitness values of other individuals belonging to other sub-populations. The interaction in terms of fitness assignment could be seen as a *competition* between the individuals because an improvement of the fitness value of a particular individual leads to the degradation of the fitness value of some others. Such competition between different solutions belonging to different species allows not only guiding the search of each sub-population towards fitter individuals but also escaping from local optima [2]. Several competitive CCEAs have been demonstrated to be effective and efficient in solving different kinds of problems such as the sorting network problem [3] and the integrated manufacturing planning and scheduling one [4]. Within the SBSE community, there are three works using competitive CCEAs such as: (1) Wilkerson et al. [5] tackling the software correction problem, (2) Arcuri and Ya [6] handling the bug fixing problem and (3) Adamopoulos et al. [7] tackling the mutation testing problem. In this paper, we present the first adaptation of *CCEA* to detect code-smells.

3.1.2 Competitive Co-evolution-Based Code-Smells Detection

The concept of co-evolution is based on the idea that two populations are evolved in parallel with a specific designed genetic algorithm (GA) [2]. The main component of our proposal is the competitive co-evolutionary algorithm. This type of co-evolution is comparable to what happens between prey and predators. Preys are the potential solutions to the optimization problem, while the predators are individuals aiming to check the survival ability of prey. In general, faster prey escape predators easily, thus they have higher chance of generating offspring. This influences the predators, since they need to evolve as well to get faster if they need to survive.

As described in Figure 1, based on this metaphor two populations evolves in parallel to reach two objectives in a competitive way. The first population uses knowledge from code-smells' examples (input) to generate detection rules based on quality metrics (input). It takes as inputs a base (i.e. a set) of code smells' examples, and takes, as controlling parameters, a set of quality metrics [21] and generates as output a set of rules. The rule generation process chooses randomly, from the metrics provided list, a combination of quality metrics (and their threshold values) to detect a specific code-smell. Consequently, a solution is a set of rules that best detect the code-smells of the base of examples. For example, the following rule states that a class *c* having more than 10 attributes and more than 20 methods is considered as a blob smell: R1: IF NAD(c)≥10 AND NMD(c)≥20 Then Blob(c). In this exemplified sample rule, the number of attributes (NAD) and the number of methods (NMD) of a class correspond to two quality metrics that are used to detect a blob. The detection rules solutions are evaluated based on the coverage of the base of code-smell examples (input) and also the coverage of generated “artificial” code-smells by the second population. These two measures are to maximize by the population of detection rules solutions.

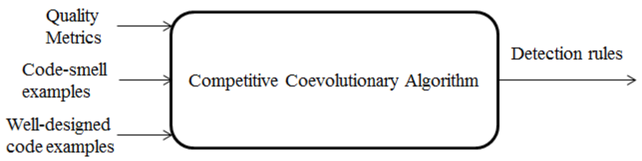
Quality Metrics
Code-smell examples
Well-designed code examples
Competitive Coevolutionary Algorithm
Detection rules

Fig. 1. Approach overview

The second population executed in parallel uses well-designed code examples to generate “artificial” code-smells based on the notion of deviation from a reference (well-designed) set of code fragments [18].The generation process of artificial code-smell examples is performed using a heuristic search that maximizes on one hand, the distance between generated code-smell examples and reference code examples and, on the other hand, minimizes the number of generated examples that are not detected by the first population (detection rules). The similarity function used is based on the distance (difference) between quality metrics [21].

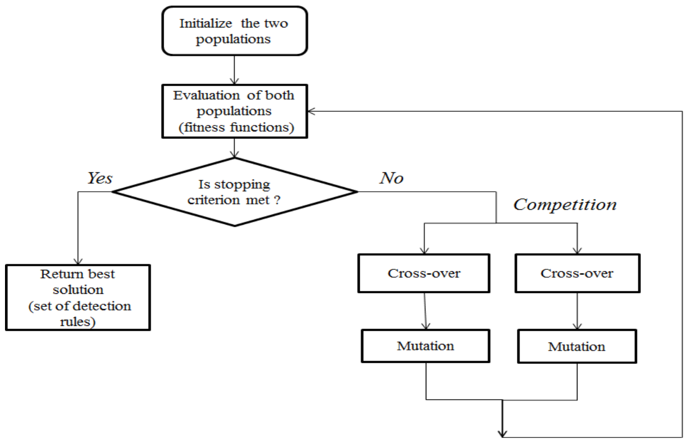
Initialize the two populations
Evaluation of both populations (fitness functions)
Yes
Is stopping criterion met ?
No
Competition
Return best solution (set of detection rules)
Cross-over
Cross-over
Mutation
Mutation

Fig. 2. Competitive co-evolutionary algorithm for code-smells detection

Figure 2 describe the overall process of CCEA for code-smells detection. The first step of the algorithm consists of generating randomly two populations. In our case, a first population generates detection rules from the list of metrics (input) and a second population generates “artificial” code-smell examples. Each population is evaluated using a fitness function. The first population maximizes the coverage of both: code-smells in the base of examples and generate “artificial” code-smell examples generated by the second population. The second population maximizes the number of generated code-smell examples that are not covered/detected by the first population and the distance with a reference (well-designed) set of code fragments. Then, change operators (selection, cross-over and mutation) are applied to generated new solutions (populations). The process is iterated until a termination criterion is met (e.g. number of iterations). Next we describe our adaptation of CCEA [2] to the test cases generation problem in more details.

3.2 Competitive Co-evolutionary Adaptation

In this section, the main contribution of the paper is presented, namely, a method for evolving test case models in parallel with mutation analysis using *CCEA*.

3.2.1 Solution Representations

For the first population that generated detection rules, a solution is composed of terminals and functions. 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 candidates solutions (rules) corresponds to a logic program that is represented as a forest of AND-OR trees.

For the second population, the generated code-smell examples represent artificial code fragments composed by code elements. Thus, these examples 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.

To generate initial populations, 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.

3.2.2 Fitness Functions

The fitness function quantifies the quality of the proposed solutions (individuals). For the first population, to evaluate detection-rules solutions the fitness function is based on: (1) maximizing the coverage of the base of code-smell examples (input) and (2) maximizing the number of covered "artificial" code-smells generated by the second population executed in parallel. For the second population, executed in parallel, to evaluate generated code-smell examples the fitness function is based on: (1) a dissimilarity score, to maximize, between generated code-smells and different reference code fragments and (2) maximizing the number of generated code-smell examples un-covered by the solutions of the first population (detection rules). In the following, we detail these functions.

The objective function of the first population checks to maximize the number of detected code-smells in comparison to the expected ones in the base of examples (input) and the generated "artificial" code-smells by the second population. In this context, we define this objective function of a particular solution *S*, normalized in the range $[0,1]$ as follows:

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

where *r* is the minimum number (among all generated detection solutions) of detected "artificial" code-smells divided by the number of generated ones (precision), *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 i^{th} component of *S* such that:

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

The second population should seek to optimize the following two objectives:

(1) Maximize the generality of the generated "artificial" code-smells by maximizing the similarity with the reference code examples;
(2) Maximize the number of un-covered "artificial" code-smells by the solutions of the first population (detection rules)

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 generated code-smells) is evaluated as the average costs of the included code-smells. Formally, the fitness function to maximize is

$$\cos t(d_i) = Max(\sum_{j=1}^{w}\sum_{k=1}^{l}\left|M_k(d_i) - M_k(c_j)\right|) + z$$

where *w* is the number of code elements (e.g. classes) in the reference code (c), *l* is the number of metrics, *M* is a metric (such as number of methods, number of attributes, etc.) and *z* is the minimum number of artificial code-smells (among all solutions) un-covered by the solutions of the first population over the number of generated “artificial” code-smells.

3.2.3 Change Operators

Selection

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 individuals 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.

Mutation

For the first population, 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 the second population, the mutation operator consists of randomly changing a predicate (code element) in the generated predicates.

Crossover

For the first population, 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 rules 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 the second population, 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 = CAMMPPP and p_2 = CMPRMPP, then o_1 = CAMRMPP and o_2 = CMPMPPP.

4 Validation

In order to evaluate our approach for detecting code-smells using *CCEA*, we conducted a set of experiments based on four large open source systems [14] [15] [16] [17].

4.1 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 [12] [18]. More specifically, we aimed at answering the following research questions (*RQ*):

- *RQ1:* To what extent can the proposed approach detect efficiently code-smells (in terms of correctness and completeness)?
- *RQ2:* To what extent does the competitive co-evolution approach performs better than the considered single-population ones?

To answer *RQ1*, we used an existing corpus [19] [13] containing an extensive study of code-smells on different open-source systems: (1) ArgoUML v0.26 [16], (2) Xerces v2.7 [15], (3) Ant-Apache v1.5 [14], and (4) Azureus v2.3.0.6 [17]. Our goal is to evaluate the correctness and the completeness of our *CCEA* code-smells detection approach. For *RQ2*, we compared our results to those produced, over 30 runs, by existing single-population approaches [12] [18]. Further details about our experimental setting are discussed in the next subsection.

4.2 Experimental Settings

Our study considers the extensive evolution of different open-source Java analyzed in the literature [13] [18] [19]. The corpus [19] [13] used includes Apache Ant [14], ArgoUML [16], Azureus [17] and Xerces-J [15]. 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 700 code-smells have been identified manually. Indeed, in [13] [18] [19], authors asked different groups of developers to analyze the libraries to tag instances of specific code-smells to validate their detection techniques. For replication purposes, they provided a corpus of describing instances of different code-smells that includes blob classes, spaghetti code, and functional decompositions. These represent different types of design risks. In our study, we verified the capacity of our approach to locate classes that correspond to instances of these code-smells.

We choose the above-mentioned open source systems because they are medium/large-sized open-source projects and were analyzed in the related work.

The initial versions of Apache Ant were known to be of poor quality, which has led to major revised versions. Xerces-J, ArgoUML, and Azureus have been actively developed over the past 10 years, and their design has not been responsible for a slowdown of their developments.

Table 1. The Systems Studied.

Systems	*Number of classes*	*Number of code-smells*
ArgoUML v0.26	1358	138
Xerces v2.7	991	82
Ant-Apache v1.5	1024	103
Azureus v2.3.0.6	1449	108

For the first population, one open source project is evaluated by using the remaining systems as a base of code-smells' examples to generate detection rules. For the second population, JHotdraw [22] was chosen as an example of reference code because it contains very few known code-smells. Thus, in our experiments, we used all the classes of JHotdraw as our example set of well-designed code.

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 (i.e., how many code-smells have not been missed).

We remove the system to evaluate from the base of code-smell examples when executing our *CCEA* 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 [12, 18]. We used precision and recall scores for all these comparisons over 51 runs. Since the used algorithms are meta-heuristics, they produce different results on every run when applied to the same problem instance. To cope with this stochastic nature, we used the Wilcoxon rank sum test [23] in the comparative study. For our experiment, we generated at each iteration up-to 150 "artificial" code-smells from deviation with JHotDraw (about a quarter of the number of reference examples) with a maximum size of 256 characters. We used the same parameter setting for *CCEA* and single-population algorithms [12, 18]. The population size is fixed to 100 and the number of generations to 1000. In this way, all algorithms perform 100000 evaluations. A maximum of 15 rules per solution and a set of 13 metrics are considered for the first population [21]. These standard parameters are widely used in the literature [2, 6, 7].

4.3 Results and Discussions

Tables 2 and 3 summarize our findings. Overall, as described in table 2, we were able to detect code-smells on the different systems with an average precision higher than 83%. For Xerces, and Ant-Apache, the precision is highest than other systems with more than 92%. This is can be explained by the fact that these systems are smaller than others and contain lower number of code-smells to detect. For ArgoUML, the precision is also high (around 90%) and most of detected code-smells are correct. This is confirms that our *CCEA* precision results are independent from the size of the systems to evaluate. For Azureus, the precision using CCEA is the lowest (71%) but still acceptable. Azureus contains a high number of spaghetti-code that are difficult to detect using metrics. For the same dataset, we can conclude that our CCEA approach performs much better (with a 99% confidence level) than existing single-population approaches (Genetic Programming and Artificial Immune Systems) [12, 18] on the different systems since the median precision scores are much higher by using *CCEA*. In fact, *CCEA* provides better results since both single-population approaches (GP and AIS) are using only manually collected examples however CCEA has the strength to generate also automatically code-smell examples during the optimization process. GP and AIS requires high number of examples to achieve good detection results.

Table 2. Precision median values of CCEA, GP, and AIS over 30 independent simulation runs

	CCEA	**GP [18]**		**AIS [12]**	
Systems	Precision	Precision	*p-value*	Precision	*p-value*
Azureusv2.3.0.6	71	62	< 0.01	65	< 0.01
Argo UMLv0.26	91	81	< 0.01	77	< 0.01
Xercesv2.7	93	84	< 0.01	83	< 0.01
Ant-Apachev1.5	93	86	< 0.01	86	< 0.01

The same statistical analysis methodology is performed to compare the recall median values. According to table 3, all median recall values of *GP/AIS* are statistically different from the *CCEA* ones on almost problem instances. Thus, it is clear that *CCEA* performs better than *GP* and *AIS*. The average recall score of *CCEA* on the different systems is around 85% (better than precision). Azureus has the lowest recall score with 74%. In fact, Azureus has the highest number of expected code-smells. Single-population approaches (GP and AIS) provide also good results (an average of 72%) but lower than CCEA ones. Overall, all the three code smell types are detected with good precision and recall scores in the different systems since the average precision and recall scores on the different systems is higher than 85%.

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 *CCEA* method can be used out of the box and this will produce good detection results for the detection of code-smells for 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.

Figures 2 shows that only code-smell examples extracted from three different open source systems can be used to obtain good precision and recall scores. In fact, since *CCEA* generates "artificial" code smell examples thus only few manually collected code-smells are required to achieve good detection results. This reduces the effort required by developers to inspect systems to produce code-smell examples.

Table 3. Recall median values of CCEA, GP, and AIS over 30 independent simulation runs

	CCEA	**GP [18]**		**AIS [12]**	
Systems	Recall	Recall	*p-value*	Recall	*p-value*
Azureus v2.3.0.6	74	62	< 0.01	66	< 0.01
ArgoUMLv 0.26	84	79	< 0.01	88	< 0.01
Xercesv2.7	88	83	< 0.01	86	< 0.01
Ant-Apachev1.5	92	80	< 0.01	84	< 0.01

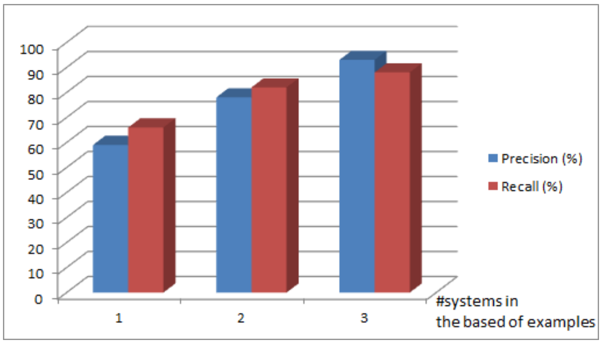

Fig. 3. The impact of number of systems in the base of examples on the detection results (Xerces)

Finally, all the algorithms under comparison were executed on machines with Intel Xeon 3 GHz processors and 8 GB RAM. We recall that all algorithms were run for 100 000 evaluations for all algorithms. This allows us to make fair comparisons in terms of CPU times. The average execution time for all the three algorithms over 30 runs is comparable with an average of 1h and 22 minutes for *CCEA*, 1h and 13 minutes for GP, and finally 1h and 4 minutes for AIS. We consider that this represents scalable results since code-smells detection algorithms are not used in real-time settings.

5 Related Work

In the literature, the first book that has been specially written for design smells was by Brown et al. [9] which provide broad-spectrum and large views on design smells, and antipatterns that aimed at a wide audience for academic community as well as in industry. Indeed, in [20], Fowler and Beck have described a list of design smells which may possibly exist on 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.

Moha et al. [19] described code-smell symptoms using a domain-specific-language (DSL) for their approach called DECOR. They proposed a consistent vocabulary and DSL to specify antipatterns based on the review of existing work on design code-smells found in the literature. Symptoms descriptions are later mapped to detection algorithms. Similarly, Munro [24] have proposed description and symptoms-based approach using a precise definition of bad smells from the informal descriptions given by the originators Fowler and Beck [20]. 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: (1) a code smell name, (2) a text-based description of its characteristics, and (3) heuristics for its detection. Marinescu [10] 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.

Our approach is inspired by contributions in the domain of Search-Based Software Engineering (SBSE) [25]. 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 [18], 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 [12], 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 [12, 18] using CCEA to detect code-smells.Based on recent SBSE surveys [25], 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. However, there is mainly three works that used *CCEA* for SBSE problems: Wilkerson et al. [5] tackling the software correction problem, Arcuri and Ya [6] handling the bug fixing problem and Adamopoulos et al. [7] tackling the mutation testing problem.

6 Conclusion

In this paper, we described a new search-based approach for code-smells detection. In our competitive co-evolutionary adaptation, two populations evolve simultaneously with the objective of each depending upon the current population of the other. The first population generates a set of detection rules that maximizes the coverage of code-smell examples and "artificial" code smells, and simultaneously a second population tries to maximize the number of "artificial" code-smells that cannot be detected by detection rules generated by the first population. We implemented our approach and evaluated it on four open-source systems. Promising results are obtained where precision and recall scores were higher than 80% on an existing benchmark [19] [13].

Future work should validate our approach with more open-source systems in order to conclude about the general applicability of our methodology. Also, in this paper, we only focused on only three types of code-smell. We are planning to extend the approach by automating the detection various other types.

References

1. Rosin, C.R., Belew, R.K.: New Methods for Competitive Coevolution. Evolutionary Computation 5(1), 1–29 (1997)
2. Stanley, K.O., Miikkulainen, R.: Competitive Coevolution through Evolutionary Complexification. Journal of Artificial Intelligence Research 21(1), 63–100 (2004)
3. Hillis, W.D.: Co-Evolving Parasites Improve Simulated Evolution as an Optimization Procedure. In: Langton, et al. (eds.) Articial Life II, pp. 313–324. Addison Wesley (1992)
4. Husbands, P.: Distributed Coevolutionary Genetic Algorithms for Multi-Criteria and Multi-Constraint Optimisation. In: Fogarty, T.C. (ed.) AISB-WS 1994. LNCS, vol. 865, pp. 150–165. Springer, Heidelberg (1994)
5. Wilkerson, J.L., Tauritz, D.R., Bridges, J.M.: Multi-objective Coevolutionary Automated Software Correction. In: GECCO 2012, pp. 1229–1236 (2012)
6. Arcuri, A., Yao, X.: Novel Co-evolutionary Approach to Automatic Software Bug Fixing. In: IEEE Congress on Evolutionary Computation, pp. 162–168 (2008)
7. Adamopoulos, K., Harman, M., Hierons, R.M.: How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-Evolution. In: Deb, K., Tari, Z. (eds.) GECCO 2004. LNCS, vol. 3103, pp. 1338–1349. Springer, Heidelberg (2004)
8. Meyer, B.: Object Oriented Software Construction, 2nd edn. Prentice Hall, New Jersey (1997)
9. Brown, W.J., Malveau, R.C., Brown, W.H., Mowbray, T.J.: Anti Patterns: Refactoring Software,Architectures, and Projects in Crisis, 1st edn. John Wiley and Sons (March 1998)
10. Marinescu, R.: Detection strategies: metrics-based rules for detecting design flaws. In: Proceedings of the ICSM 2004, pp. 350–359 (2004)
11. Fenton, N., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach, 2nd edn. International Thomson Computer Press, London (1997)
12. Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., Ouni, A.: Design Defects Detection and Correction by Example. In: 19th ICPC 2011, Canada, pp. 81–90 (2011)

13. Ouni, A., Kessentini, M., Sahraoui, H., Boukadoum, M.: Maintainability Defects Detection and Correction: A Multi-Objective Approach. In: Journal of Automated Software Engineering (JASE). Springer (2012)
14. `http://ant.apache.org/`
15. `http://xerces.apache.org/xerces-j/`
16. `http://argouml.tigris.org/`
17. `http://sourceforge.net/projects/azureus/`
18. Kessentini, M., Vaucher, S., Sahraoui, H.: Deviance from Perfection is a Better Criterion than Closeness to Evil when Identifying Risky Code. In: 25th IEEE/ACM ASE 2010 (2010)
19. Moha, N., Guéhéneuc, Y.-G., Duchien, L., Le Meur, A.-F.: DECOR: A Method for the Specification and Detection of Code and Design Smells. TSE 36, 20–36 (2010)
20. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring – Improving the Design of Existing Code, 1st edn. Addison-Wesley (1999)
21. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object-oriented design. IEEE Trans. Softw. Eng. 20(6), 293–318 (1994)
22. `http://www.jhotdraw.org/`
23. Wilcoxon, F., Katti, S.K., Roberta, A.: Critical Values and Probability Levels for the Wilcoxon Rank Sum Test and the Wilcoxon Signed-rank Test. In: Selected Tables in Mathematical Statistics, vol. I, pp. 171–259. American Mathematical Society (1973)
24. Munro, M.J.: Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In: 11th METRICS Symp. (2005)
25. Harman, M., Afshin Mansouri, S., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv. 45, 61 pages (2012)