Software Refactoring Using Cooperative Parallel Evolutionary Algorithms

Troh Josselin Dea, Marouane Kessentini, William I. Grosky
Department of Computer and Information Science
University of Michigan, USA
firstname@umich.edu

Kalyanmoy Deb
Department of Computer Science and Engineering
Michigan State University, USA
kdeb@egr.msu.edu

COIN Report Number 2016021

Abstract— **Code bad smells represent symptoms of poor implementation choices. Previous studies found that these smells make source code more difficult to maintain, possibly also increasing its fault-proneness. There are plethora of approaches that identify bad smells based on code analysis techniques. We propose, in this paper, to consider code-smells correction as a distributed problem. Different techniques are combined during an optimization process to find a consensus regarding the correction of code-smells. To this end, we used Distributed Evolutionary algorithms (D-EA) where many evolutionary algorithms with different adaptations (fitness functions, solution representation, and change operators) are executed in parallel to solve a common goal which is the correction of code-smells. The statistical analysis of the obtained results provide evidence to support the claim that cooperative D-EA outperforms single population evolution and random search based on a benchmark of eight large open source systems where more than 86% of code-smells are fixed using the suggested refactorings.**

Keywords—Search-based software engineering, applications, software refactoring, evolutionary algorithms.

I. INTRODUCTION

Nowadays, maintenance and evolution are critical software development activities that might comprise up to 80% of the overall cost and effort [13]. Maintenance costs often can be inflated through poor software design called code-smells and defined as design situations that adversely affect the software maintenance [12]. One of the widely used techniques to fix code-smells is refactoring defined as the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [20].

Recently, refactoring has been gaining widespread acceptance from both academia and industry. Some work proposes "standard" refactoring solutions that can be applied by hand for each kind of code-smells (e.g. blobs, large methods, etc.). However, there is no consensus to define these standard solutions since the same type of code-smell can be fixed by many alternatives. In fact, developers can propose different refactoring solutions to fix the same code fragment and these solutions have different adaptability/calibration effort. Thus, the question is how to find a consensus between all these possible solutions? Some researchers start from the hypothesis that useful refactorings are those which improve the metrics [20][21]. Some limitations can be drawn from studying these metric-based approaches. First, how to determine the useful metrics for a given system? Second, how best to combine multiple metrics? Third, improving the metrics values do not means that specific defects are corrected. Furthermore, it is difficult to find a consensus regarding the definition of thresholds values for the used quality metrics.

To address these issues and mainly due to the absence of consensus to find the best sequence of refactoring that fix code-smells, we propose in this paper to consider code-smells correction as a distributed problem. The idea is that different techniques are combined during an optimization process to find a consensus regarding the correction of code-smells. To this end, we used Distributed Evolutionary algorithms (D-EA) [2][8] where many evolutionary algorithms with different adaptations (fitness functions, solution representation, and change operators) are executed in parallel to solve a common goal which is the correction of code-smells. We believe that a D-EA approach is suitable to our problem because it combines between different expertises (algorithms) to correct code-smells. We show how this combination can be formulated in a cooperative search. In D-EA, many refactoring algorithms evolve simultaneously where the goal is to maximize the intersection between the refactoring solutions proposed by different evolutionary algorithms (with different adaptations) while satisfying also their own fitness functions. We executed in parallel two algorithms that we proposed in our previous work [16][23]. We implemented our approach and evaluated it on eight open source systems using an existing benchmark. We report the results on the efficiency and effectiveness of our approach, compared to a random search in addition to two different existing single population approaches [16][23]. Our results indicate that the D-EA approach has great promise. Over 51 runs for each approach, D-EA significantly outperforms both random and single population approaches in terms of the percentage of fixed code-smells by the suggested refactoring. The first algorithm is based on genetic algorithm to find the best sequence of refactorings that can minimize the number of code-smells detected using metrics-based rules [16]. The second algorithm is based also on genetic algorithm to find the best sequence of refactorings that can minimize the distance between the code-fragments to refactor (code-smells) and a set well-designed code examples [23]. The primary contributions of this paper can be summarized as follows: (1)

the paper introduces a novel formulation of the code-smells' fixing problems as a distributed problem, to the best of our knowledge, this is the first paper in the literature to use distributed evolutionary algorithms to software refactorings; (2) the paper reports the results of an empirical study with an implementation of our D-EA approach, compared to random search and two existing single population approaches [16][23] (the two used GA algorithms executed separately). The statistical analysis of the obtained results over 51 runs shows that D-EA is more efficient and effective than single population evolution and random search based on a benchmark of eight large open source systems.

II. Software Refactoring Overview

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

The detected defects can be fixed by applying some refactoring operations. For example, to correct the blob code-smell (large class containing most of the functionalities of the system) many operations can be used to reduce the number of functionalities in a specific class: move methods, extract class, etc. Fowler [12] defines refactoring as the process of improving a code after it has been written by changing the internal structure of the code without changing the external behavior. The idea is to reorganize variables, classes and methods in order to facilitate future extensions. This reorganization is used to improve different aspects of software-quality: reusability, maintainability, complexity, etc [13]. Some examples of refactoring operations include: (1) Push down field: moves a field from some class to those subclasses that require it, (2) Add parameter: adds a new parameter to a method, (3) Push down method: moves a method from some class to those subclasses that require it, and (4) Move method: moves a method from one class to another. Overall, the refactoring process consists of three main steps [12]: (1) identify where the software should be refactored (e.g. bad-smells detections); (2) select which refactorings should be applied to the identified code fragments; and (3) ensure that the applied refactoring preserves the program behavior.

Some issues need to be address when automating code-smells correction using refactoring. In many situations, code quality can be improved without fixing code-smells. We need to identify if the code modification corrects or not some specific code-smells. In addition, the code quality is estimated using quality metrics but different problems are related to how to determine the useful metrics for a given system and how to combine in the best way multiple metrics to detect or correct code-smells. As mentioned earlier, it is difficult to find a consensus to define thresholds values for metrics. In addition, specifying manually a "standard" refactoring solution for each type of code-smell can be a difficult task due to the large list of possible defects and possible fixing solutions. Indeed, these "standard" solutions can remove all symptoms for each defect. However, removing the symptoms does not mean that the defect is corrected. Moreover, the process of defining rules manually for detection or correction is complex (absence of consensus to define "standard" refactoring solutions), time-consuming and error-prone. Another important issue is that there is no consensus to define these standard solutions since the same type of code-smell can be fixed by many alternatives. In fact, developers can propose different refactoring solutions to fix the same code fragment and these solutions have different adaptability/calibration effort. Thus, the question is how to find a consensus between all these possible solutions? To answer this question, we believe that it is important to take the best from different refactoring techniques that can be applied to fix code-smells using a distributed approach that is described in the next section.

III. Code-Smells Correction Using Cooperative Parallel Evolutionary Algorithms

This section shows how the above-mentioned issues can be addressed using our proposal and describes the principles that underlie the proposed method for fixing code-smells. Therefore, we first present an overview of the used distributed evolutionary (D-EA) algorithms and, subsequently, provide the details of the approach and our adaptation of the D-EA to fix code-smells.

A. *Parallel Evolutionary Algorithms*

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 (EAs) 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 have motivated researchers to more focus on designing and implementing parallel metaheuristics 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 parallel metaheuristics. Firstly, parallelization permits to speed up the search process by reducing the search time. This is very interesting in time-dependent and interactive resolution methods. Secondly, the quality of the obtained solutions may be significantly improved. In fact, cooperative metaheuristics has been demonstrated to explore the fitness landscape more efficiently on different problems such as the defect 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 efficiently. Thirdly, the use of different metaheuristics simultaneously in solving a particular 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 *robustness* of the obtained results. Finally, parallel distributed metaheuristics allows tackling the problematic of scalability. Several problems actually involves 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. Parallel distributed metaheuristics can represent one possible remedy to tackle such problems.

Different models exist for designing parallel metaheuristics. According to [7], these models follow the following three hierarchical levels:

- *Algorithmic level*: In this parallel model, independent or cooperating self-contained metaheuristics are used. It can be seen as a *problem-independent inter-algorithm parallelization*. If the different metaheuristics are independent, the search will be equivalent to the sequential execution of the metaheuristics in terms of the quality of solutions. However, the cooperative model will alter the behavior of the metaheuristics and enable the improvement of the quality of solutions.
- *Iteration level*: In this model, metaheuristic iterations are parallelized. It is *a problem-independent intra-algorithm parallelization*. The metaheuristic behavior is not altered. The main goal is to speed up the algorithm by reducing the search time. In fact, the iteration cycle of metaheuristics on large neighborhoods for trajectory-based metaheuristics or large populations for population-based metaheuristics 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 metaheuristics. In this model, the metaheuristic behavior is not altered. The goal is mainly the speed up of the search.

Although parallel distributed EAs' is still a challenging research field, several contributions are mature enough today 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 D-EAs to solve software engineering problems, and particularly the code-smells correction one. Indeed, to the best of our knowledge and based on recent surveys [22], D-EAs are applied in software engineering only to the test cases generation problem by Alba and Chicano [24]. Thus, this work represents the first research contribution to solve the code smells detection problem using D-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% elite solutions, i.e., those with the highest fitness values, are updated according to a *parallel fitness function*. The update operation role is to penalize solutions that do not perform well according to the parallel model and to favour good solutions w.r.t. the parallelization. For the problem considered in this work, code-smells correction, we use a simple *bi-algorithm parallel* model illustrated by figure 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 parallel fitness-update operation is alleviating the encountered challenges of the code-smell correction, which are discussed in the previous section. From a semantic viewpoint, the parallel model goal is *to circumvent the absence* of a well-defined consensus concerning the correction of code-smells (symptoms, thresholds value, adaptability effort, semantic, etc.) where the goal is to maximize the intersection between the different refactoring solutions. The next section describes in details our adaptation of D-EA.

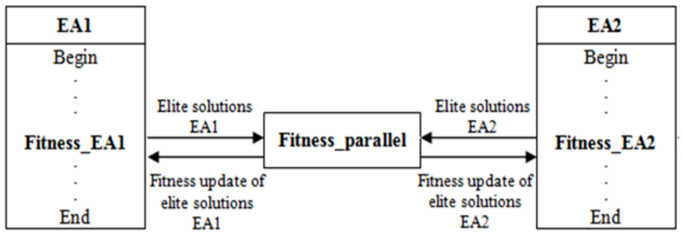

Fig.1 The proposed cooperative parallel model scheme

B. *Distributed Code-Smells Correction : Overview*

To circumvent the above mentioned issues, we propose a framework based on the use of two evolutionary algorithms in parallel with completely different adaptations [16][23]. The two algorithms interact during the optimization process by maximizing the intersection (in terms of number of common refactorings) between the best refactoring solutions provided by both algorithms for the system to evaluate/improve.

The general structure of our approach is introduced in Figure 2. We are using two different evolutionary algorithms from our previous work [16][23] to find a consensus to correct code-smells.

The first algorithm is based on genetic algorithm [16] to find the best sequence of refactorings that can minimize the number of code-smells detected using metrics-based detection rules generated by a genetic programming algorithm. It takes as controlling parameters, the detection rules, and a set of refactoring operations that were defined and discussed in the related literature [12]. It takes as input a system (*A*) containing code-smells to be fixed. As output, this step recommends a refactoring solution, which suggests a set of refactoring operations that should be applied in order to correct the input code. The algorithm start by using the detection rules to detect code-smells in the input code (*A*). Then the process of generating a correction solution can be viewed as the mechanism that finds the best way to combine some subset (to be fixed during the search) among all available refactoring operations, in such a way to best reduce the number of detected code-smells. More details about this algorithm can be found in [16].

The second algorithm is based on genetic algorithm [23] also to find the best sequence of refactorings that can minimize the distance between the well-designed code examples and the system (*A*) (after applying the refactorings solution) using alignment techniques [23]. This algorithm takes as controlling parameters examples of well-designed (reference) code, and a set of refactoring types. It takes as input the list of code-smells detected on system (*A*) to correct, and the well-designed code examples. As output, this algorithm recommends a refactoring solution, which suggests a set of refactoring operations that should be applied to improve the quality of the input code. The process of generating a correction solution is the mechanism that finds the best way to combine some subset (to be fixed during the search) among all available refactoring operations, in such a way as to best minimizes the similarity between the reference code and code fragments containing bad-smells. The similarity distance is also based on the use of global and local alignment techniques. A detailed description of this algorithm can be found in [23].

In our D-EA framework, both algorithms are executed in parallel and during the optimization process (each iteration) they interact by maximizing the intersection between the best solutions of suggested refactorings, by both algorithms, to correct the system (*A*). In fact, at each iteration, our D-EA selects the best solutions from both algorithms using a selection operator. Then, the intersection score between the suggested refactorings is to maximize. This score is used to penalize or favor the best solutions of both algorithms.

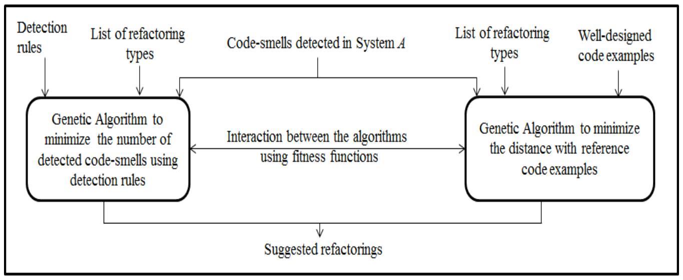
Detection rules
List of refactoring types
Code-smells detected in System A
List of refactoring types
Well-designed code examples
Genetic Algorithm to minimize the number of detected code-smells using detection rules
Interaction between the algorithms using fitness functions
Genetic Algorithm to minimize the distance with reference code examples
Suggested refactorings

Fig 2. Distributed Evolutionary Algorithms for code-smells correction

C. *Distributed Code-Smells Correction : Problem Formulation*

This subsection is devoted to describe the formal formulation of the code-smells correction problem using D-EA. 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.

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

Lines 1–3 construct initial GA populations, which are sets of individuals that stand for possible solutions representing a random combination of refactoring operations. Lines 4–15 encode the main loops for both algorithms, which explores the search space and constructs new individuals by combining refactorings. At each iteration, we evaluate the quality of each individual in the population for each algorithm (lines 6 and 13). The first algorithm evaluates the refactoring solutions on the number of detected code-smells using detection rules and the second algorithm evaluates solutions based on the the deviance from well-designed code (input) using global and local alignment techniques. Then, a set of best solutions are selected form both algorithms in each iteration. Both algorithms interact with each other using *intersection_fitness* where the goal is to maximize the *intersection* between the sets of suggested refactoring operations by each solution. Thus, some solutions will be *penalized* due to the absence of consensus and others will be favored (lines 11-16). 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. Both algorithms terminates when the termination criterion (maximum iteration number) is met, and returns the best set of refactorings solution from both algorithms. In the following, we describe the three main steps of adaptation for both algorithms GP and GA to our problem.

We used the same solution representation and change operators used in our previous work [16][23]. Thus, the solutions of both algorithms are represented as a vector where each dimension is a refactoring operation (e.g. move method, add attribute, etc.) applied to some code elements of system *A* to fix. For instance, the solution represented in Figure 3 is composed of three steps corresponding to three refactoring operations to apply in some code fragments.

For both algorithms, after selecting parents using the roulette wheel selection mechanism, the mutation operator consists of randomly replacing some dimensions in the vector (refactoring) with new ones. The crossover operation creates two offspring I'1 and I'2 from the two selected parents I1 and I2. It is defined as follows: A random position, *k*, is selected. The first *k* refactorings of I1 become the first *k* elements of I'1. Similarly, the first *k* refactorings of I2 become the first k refactorings of I'2.

One of the unique aspects of our distributed evolutionary algorithm is the use of an intersection fitness function. In fact, a set of best solutions are selected (5% of the population) from both algorithms, in each iteration, and then executed on a new system *A* to fix. A matrix is constructed where rows are composed by best solutions of the first GA {SR_i}, columns are composed by best solution of second GA {SD_j} and each case (SR_i, SD_j) is defined as:

$$f_{intersection}(SR_i, SD_j) = \frac{\left|\{\text{refactoring of } \text{SR}_\text{i}\} \cap \{\text{refactoring of } \text{SD}_\text{j}\}\right|}{\text{Max}\left(\left|\text{SR}_\text{i}\right|, \left|\text{SD}_\text{j}\right|\right)}$$

Then, the intersection score for each solution is defined as:

$$f_{intersection}(SR_i) = Max(f_{intersection}(SR_i, SD_{\forall j}))$$

and

$$f_{intersection}(SD_j) = Max(f_{intersection}(SR_{\forall i}, SD_j))$$

In this case, the fitness function, to maximize, used by the first algorithm is defined as

$$f_1 = f_{interstion} - n,$$

where n is the number of code-smells detected by the detection rules after applying the solution (refactorings) on the system A to fix. $f_{intersection}$ takes the value 0 if the solution is not selected for intersection. The fitness function, to maximize, of the second GA algorithm is defined as

$$f_2 = AlignmentSimilarity(A, GE) + f_{intersection},$$

where *AlignmentSimiliarty* score is obtained from the global alignment algorithm executed on the system A (after applying refactoring) and the set of reference systems (GE) . More details can be found about this similarity function in our previous work [23].

IV. Evaluation

In this section, we start by presenting our research questions. Then, we describe and discuss the obtained results.

A. Goals

The goal of the study is to evaluate the efficiency of our approach for correction of bad-smells from the perspective of a software maintainer conducting a quality audit. We present the results of the experiment aimed at answering the following research questions: *RQ1*: To what extent can the proposed approach generate valid refactoring sequences?; *RQ2*: To what extent can the proposed approach correct bad-smells?; *RQ3:* To what extent can D-EA performs comparing to single-population approaches [16][23](both algorithms executed separately) and random search ?

To answer *RQ1*, we manually verified the feasibility of the different proposed refactoring sequences for each system. To this end, we asked a group of experts composed of seven PhD students from University of Michigan to perform a manual validation of the suggested refactoring. All these PhD students are expert in refactoring and familiar with the used open source systems. We report the percentage of proposed refactorings that were semantically correct (did not violate the behavior of the programs): *SP* is defined as

$$Semantic - precision = \frac{\text{number of valid refactoring}}{\text{number of refactoring}}$$

```
Input: List of refactoring types RT.
Input: Detection rules DR.
Input: System A containing a list of code-smells.
Output: refactoring sequence RS.
1: I1:= refactorings(RT);
2: P1:= set_of(I1);
3: initial_populationGA(P1, Max_size) ;
4: repeat
5:        for all I1 ∈ P 1 do
6:             detected_codesmells_GA(A) := execute_rules(I1, A);
7:             fitness(I1) := number_of_code_smells(detected_codesmells_GA(A) );
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:            fitness_intersection (I1) := := Max_intersection(refactoring(I1) ∩ refactoring(best_sol_P2))/(Max_number_refactoring(I1, best_sol_P2));
14:            fitness(I1) := updatefitness (I1, fitness_intersection);
15:       end for
16:       best_solution_P1 := best_fitness(P1);
17:       P1 := generate_new_population(P1);
18:       it:=it+1;
19: until it=max_it
20: return best_solution_refactorings
```

(a)

```
Input: Set of well-designed code examples GE.
Input: System A containing a list of code-smells.
Input: List of refactoring types RT.
Output: refactoring sequence RS.
1: I2 := refactorings(RT);
2: P2= set_of(I2);
3: initial_populationGA(P2, Max_size) ;
4: repeat
5:        for all I2 ∈P2 do
6:          A := ApplyRefactoring(I2, A);
7:          fitness(I2) := distance(A,   GE);
8:        end for
9:        best_sol_P2 := select(P2, number_solutions);
10:       send(best_sol_P2);
11:       best_sol_P1 := receive(best_sol_P1);
12:       for all I2 ∈best_sol_P2 do
13:          fitness_intersection (I2) := Max_intersection(refactoring(I2) ∩ refactoring(best_sol_P1))/(Max_number_refactoring(I2, best_sol_
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_refactorings
```

(b)

Fig 3. High-level pseudo-code of our D-EA adaptation. Both algorithms interact with each other using *"fitness_intersection"* function in each interaction.

MoveMethod(division, Department, University)	AddParameter(x, int, average, Student)	PushDownMethod(average, Student, Person)

Fig 4. Individual representation for both algorithms

To answer RQ2, we used an existing corpus of known bad-smells [26] and we validated manually whether the proposed corrections were useful to fix bad-smells found in the corpus. The precision denotes the fraction of fixed code-smells among the set of all code-smells: *EP* is defined as

$$refactoring - precision = \frac{\text{number of fixed code - smells}}{\text{number of code - smells}}$$

For the remaining question, we compared our results to those produced, over 51 runs, by existing single-population approaches [16][23] in addition to random search. Further details about our experimental setting are discussed in the next subsection.

B. Studied Systems and Setting

Our study considers the extensive evolution of different open-source Java analyzed in the literature [25][26]. The corpus [25][26] used includes the releases of Apache Ant, ArgoUML, Gantt, Azureus and Xerces-J. Table I reports the size in terms of classes of the analyzed systems. The considered existing corpus contains the three well-known code-smells (Blob, Spaghetti Code, and Functional Decomposition).

We choose the above-mentioned open source systems because they are medium/large-sized open-source projects and were analyzed in the related work. JHotdraw was chosen as an example of reference code because it contains very few known code-smells. After executing our D-EA technique, we select the best refactoring solutions of both algorithms to analyze (having the better fitness score which is normalized for our experiments between 0 and 1). We used the above-described measures (*SP* and *RP*) 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, the use of rigorous statistical testing is then essential to provide support to the conclusions derived from analyzing such data. Thus, we used the Wilcoxon rank sum test in the comparative study. For the single population algorithms (GA) and random search, population size is fixed at 100 and the number of generations at 1000. For D-EA, the population size is 100 and number of generations 500. In this way, all algorithms perform 100000 evaluations (fair comparison).

Table I The Systems Studied.

Systems	*Number of classes*	*Number of code-smells*
ArgoUML v0.26	1358	138
ArgoUML v0.3	1409	129
Xerces v2.7	991	82
Ant-Apache v1.5	1024	103
Ant-Apache v1.7.0	1839	124
Gantt v1.10.2	245	51
Azureus v2.3.0.6	1449	108

C. Results and Discussions

As showed in Table 2, the majority of proposed refactoring are feasible (preserve the behavior) and improve the code quality. For instance, for Azureus, 84% of suggested refactoring are valid and fix 88% of design defects. Around 92% of suggested refatorings are correct for Gantt. The best results were obtained for Xerces-J. The main reason is that this system is smaller than the others. In addition, Xerces-J contains fewer bad-smells than the remaining systems. The average of our D-EA approach is more than 85% of SP and RP on the different eight systems. We found that the unfeasible (invalid) refactorings are mainly related to some semantic incoherence (for example when moving some methods between classes) or conflicts between the suggested refactoring. Other suggestions represent some bad recommendations. These bad recommendations correspond to different proposed code-changes related to classes that are not classified as bad-smells in the corpus. To conclude, after manually applying the feasible refactoring operations for all systems, we found that on average are better than both single population algorithms executed separately (GA1 is based on minimizing the number of code-smells [16] and GA2 is based on maximizing similarity with reference code [25]). The majority of unfixed bad-smells are blobs. In fact, this type of bad-smell needs a large number of refactoring operations and is very difficult to correct.

Table II. Precision (SP an RP) median values of D-EA, GA1, GA2 and RS over 51 independent simulation runs.

	D-EA		**GA1 [16]**		**GA2 [23]**		**RS**	
Systems	*SP*	*RP*	*SP*	*RP*	*SP*	*RP*	*SP*	*RP*
Azureus v2.3.0.6	84	88	77	82	74	71	37	19
Gantt	92	90	84	76	71	73	31	22
Argo UML (V 0.26)	81	86	73	79	68	77	27	17
Argo UML (V 0.3)	84	91	78	84	82	71	24	24
Xerces (V2.3)	93	86	82	81	84	82	27	17
Xerces (V2.7)	86	92	74	78	79	72	29	19
AntApache (V 1.5)	91	84	82	77	83	78	31	23
AntApache (V 1.7.0)	87	94	79	81	81	81	33	21

We note that the SP and RP median values of D-EA, GA1, GA2 and RS over 51 independent simulation runs. The p-values of the Wolcoxon rank sum test indicate whether the median of the algorithm of the corresponding column (GA1/GA2/RS) is statistically different from the PEA one with a 99% confidence level (α = 0.01). A statistical difference, in terms of the obtained recall values, is detected when the p-value is less than or equal to 0.01.

Figure 5 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%. For this reason, we considered this threshold in our experiments. Our D-EA technique requires a number of comparisons between the selected solutions, thus the execution time need to be considered (number of comparison). Indeed, we believe that 5% is a good threshold value for a population of 100 individuals to keep reasonable execution time.

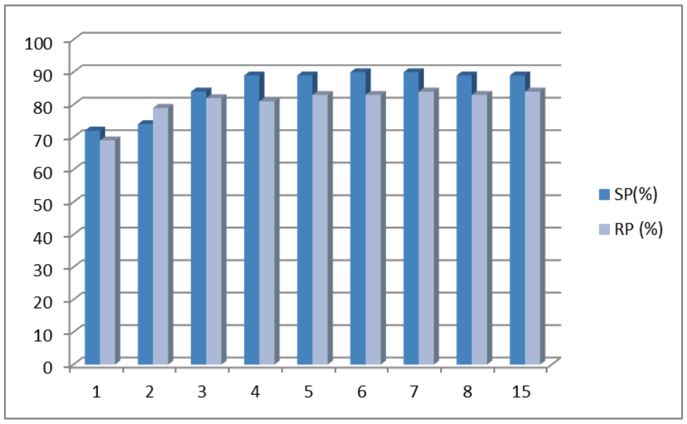

Fig 5. The impact of percentage of best solutions selected for the intersection process on detection results (average precision scores on all systems to evaluate)

All the algorithms under comparison were executed on machines with Intel Xeon 3 GHz processors and 8 GB RAM. We note that each of GA1 and GA2 were run on a single machine. However, our D-EA was executed on two nodes (machines) following the previously described parallel model. We recall that all algorithms were run for 100000 evaluations. This allows us to make a fair comparison between CPU times. Figure 6 illustrates the obtained CPU times of all algorithms on each of the considered systems. 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 D-EA is demonstrated to be faster than GA1 and GA2 as highlighted through figure 6. The D-EA spends approximately the half amount of time required for GA1 or GA2. This observation could be explained by the fact that the performed 100000 evaluations are distributed between the two machines (50000 for each). In this way, the D-EA is able to evaluate 200 individuals at each iteration, which is not the case for GA1 or GA2 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 refactoring problem.

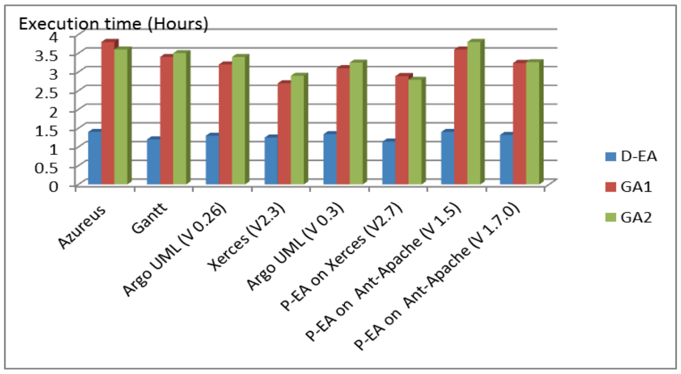

Fig 6. Average execution time on the different systems

V. Related Work

The majority of existing approaches to automate refactoring activities are based on rules that can be expressed as assertions (invariants, pre- and post-condition), or graph transformation. The use of invariants has been proposed to detect parts of programs that require refactoring by [11]. Opdyke [12] suggests the use of pre- and postcondition with invariants to preserve the behavior of the software. All these conditions could be expressed in the form of rules. Authors in [13] consider refactorings activities as graph-based rules (programs expressed as graphs). However, a full specification of refactorings could require large numbers of rules. In addition, refactoring-rules sets have to be complete, consistent, non-redundant, and correct. In such situations, search-based techniques represent a good alternative. Van belle et al. [14] presented a refactoring approach, based on genetic programming, with a periodically changing fitness function. However, the approach was not tested on large systems or to correct specific bad-smells. In [15], we have proposed another approach, based on search-based techniques, for the automatic detection of potential bad-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 [16], 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.

In [17], metrics are used as indicators for automatically detecting situations where a particular transformation can be applied to improve the quality of systems (maintainability, reliability, reusability). Suggested transformations (called prescriptions) are similar to refactorings (create an abstract class, create an aggregate class and create a specialized subclasses) and plays on the values of certain metrics. Tahvildari et al. [18] adopt the same approach and proposes a reengineering framework to improve design quality of an object oriented system using meta-pattern transformations. Design flaws and deteriorated classes are detected by quality heuristics rules. Metrics are not indicators of bug but system entities (classes and methods) with high metrics can be good candidates for refactoring. The correction step is based on the selection and application of potential transformations that can correct design flaws. In another category of work, bad-smells are not detected explicitly. They are implicitly because the approaches refactor a system by detecting elements to change to improve the global quality. For example, in [20], bad-smell detection is considered as an optimization problem. The authors use a combination of 12 metrics to measure the improvements achieved when sequences of simple refactorings are applied, such as moving methods between classes. The goal of the optimization is to determine the sequence that maximizes a function, which captures the variations of a set of metrics [21]. To conclude, to the best of our knowledge and based on recent SBSE surveys [22], the use of parallel metaheuristic search is still very limited in software engineering. Indeed, parallel metaheuristic search was applied only the problem of software testing to generate test-cases and there is no work that uses cooperative distributed metaheuristic search to software refactoring.

VI. Conclusion

In this paper, we proposed a new representation of the problem of software refactoring. In our cooperative distributed metaheuristic adaptation, two populations evolve simultaneously with the objective of each depending upon the current population of the other in a cooperative manner. Both populations are executed, on the same system to fix code-smells, and the solutions are penalized based on the consensus found, i.e.: intersection between the refactoring solution of both algorithms. The statistical analysis of the obtained results provide evidence to support the claim that cooperative D-EA outperforms single population evolution and random search based on a benchmark of eight large open source systems. Future work should validate our distributed metaheuristic approach with new software engineering problems such as project management, software evolution, etc. We are planning also to provide a distributed search-based framework that can be used in all life-cycle activities.

References

[1] Patrick Siarry, Zbigniew Michalewicz (2008) Advances in metaheuristics for hard optimization. Natural Computing Series, Springer, ISBN: 978-3-540-72959-4.

[2] Enrique Alba (2005) Parallel Metaheuristics: New Class of Algorithms, John Wiley & Sons, ISBN: 0-471-67806-6.

[3] Tadeusz Burczyńskia, Wacław Kuśa, Adam Długosza, Piotr Oranteka (2004) Optimization and defect identification using distributed evolutionary algorithms. Engineering Applications of Artificial Intelligence, 4(17): 337–344.

[4] Xiaodong Li, Xin Yao (2012) Cooperatively coevolving particle swarms for large scale optimization. IEEE Transactions on Evolutionary Computation, 16(2): 210–224.

[5] Evan J. Hughes (2005) Evolutionary many-objective optimization: Many once or one many? In: Proceedings of IEEE Congress Evolutionary Computation (CEC'05), pp. 222–227.

[6] Yong Wang, Zixing Cai, Guanqi Guo, Yuren Zhou (2007) Multiobjective optimization and hybrid evolutionary algorithm to solve constrained optimization problems. IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, 37(3): 560–575.

[7] El-Ghazali Talbi (2009) Metaheuristics - From design to implementation. John Wiley & Sons, ISBN: 978-0-470-27858-1.

[8] Carolina Salto, Enrique Alba (2012) Designing heterogeneous distributed GAs by efficiently self-adapting the migration period. Applied Intelligence, 36(4): 800–808.

[9] Marco Tomassini, Leonardo Vanneschi (2010) Guest editorial: special issue on parallel and distributed evolutionary algorithms, part two. GPEM, , 11(2): 129–130.

[10] David E. Goldberg (1989) Genetic algorithms in search, optimization and machine learning. Addison Wesley, ISBN 0-201-15767-5.

[11] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin, "Automated support for program refactoring using invariants," in Proc. ICSM. 2001, pp. 736–743.

[12] W. F. Opdyke, Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.

[13] Reiko Heckel, "Algebraic graph transformations with application conditions," M.S. thesis, TU Berlin, 1995

[14] Terry Van Belle and David H. Ackley. 2002. Code Factoring And The Evolution Of Evolvability. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '02). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1383-1390.

[15] Kessentini, M., Vaucher, S., and Sahraoui, H.:. Deviance from perfection is a better criterion than closeness to evil when identifying risky code, in Proc. of the International Conference on Automated Software Engineering. ASE'10, 2010.

[16] M. Kessentini, W.Kessentini, H.Sahraoui, M.Boukadoum and A.Ouni Design Defects Detection and Correction by Example. In Proc.ICPC 2011, pp. 81-90 IEEE, (2011)

[17] Houari A. Sahraoui, Robert Godin, and Thierry Miceli. Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In ICSM '00 : 154.

[18] Ladan Tahvildari and Kostas Kontogiannis. Improving design quality using meta-pattern transformations: a metric-based approach. Journal of Software Maintenance, 331–361

[19] Scott Grant and James R. Cordy. An interactive interface for refactoring using source transformation. In Proceedings of the First InternationalWorkshop on Refactoring: Achievements, Challenges, Effects (REFACE'03), pages 30–33, November 2003.

[20] M. O'Keeffe and M. Cinnéide: Search-based refactoring: an empirical study, Journal of Software Maintenance, vol. 20, no. 5, pp. 345–364, 2008.

[21] M. Harman and J. A. Clark: Metrics are fitness functions too. in IEEE METRICS. IEEE Computer Society, 2004, pp. 58–69.

[22] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv. 45, 61 pages.

[23] M.Kessentini, R.Mahouachi, K.Ghedira, What you like in design use to correct defects, Software Quality Journal, to appear (http://link.springer.com/content/pdf/10.1007%2Fs11219-012-9187-6)

[24] Enrique Alba, J. Francisco Chicano: Observations in using parallel and sequential evolutionary algorithms for automatic software testing. Computers & OR 353161-3183, 2008

[25] Ouni, A., Kessentini, M., Sahraoui, H., and Boukadoum, M.: Maintainability Defects Detection and Correction: A Multi-Objective Approach, in Journal of Automated Software Engineering (JASE), Springer, 2012.

[26] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. Le Meur: DECOR: A Method for the Specification and Detection of Code and Design Smells. TSE, 36, 20-36, 2010.