Software Qual J (2017) 25:529–552
DOI 10.1007/s11219-016-9309-7

Multi-objective code-smells detection using good and bad design examples

Usman Mansoor[1] · Marouane Kessentini[1] · Bruce R. Maxim[1] · Kalyanmoy Deb[2]

Published online: 15 February 2016

Abstract Code-smells are identified, in general, by using a set of detection rules. These 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. We propose in this work to consider the problem of code-smell detection as a multi-objective problem where examples of code-smells and well-designed code are used to generate detection rules. To this end, we use multi-objective genetic programming (MOGP) to find the best combination of metrics that maximizes the detection of code-smell examples and minimizes the detection of well-designed code examples. We evaluated our proposal on seven large open-source systems and found that, on average, most of the different five code-smell types were detected with an average of 87 % of precision and 92 % of recall. Statistical analysis of our experiments over 51 runs shows that MOGP performed significantly better than state-of-the-art code-smell detectors.

Keywords Search-based software engineering · Software maintenance · Software metrics

1 Introduction

Software maintenance is considered the most expensive activity in the software lifecycle (Abreu et al. 1995). Maintenance tasks can be seen as incremental modifications to a software system that aim to add or adjust some functionality or to correct some design flaws and fix some bugs. It has been found that feature addition, modification, bug fixing,

✉ Marouane Kessentini
marouane@umich.edu

Kalyanmoy Deb
kdeb@egr.msu.edu

[1] University of Michigan, Dearborn, MI, USA

[2] Michigan State University, East Lansing, MI, USA

and design improvement can cost as much as 80 % of total software development cost (Travassos et al. 1999). In addition, it is shown that software maintainers spend around 60 % of their time in understanding the code (Zitzler et al. 2003). This high cost could potentially be greatly reduced by providing automatic or semiautomatic 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 code-smells also called design defects or anti-patterns or anomalies in the literature (Yamashita and Moonen 2012, 2013; Brown et al. 1998; Kessentini et al. 2011a, b; Mäntylä and Lassenius 2006; Moha et al. 2010; Palomba et al. 2014; Sahin et al. 2014). Although these defects 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, these code-smells refer to design situations that may adversely affect the maintenance of software. They make a system difficult to change, which may in turn introduce bugs. In this paper, we focus on the detection of code-smells.

Several studies were proposed in the literature of code-smell detection based, for example, on declarative rule specification (Brown et al. 1998; Sjøberg et al. 2013; Van Emden and Moonen 2002; Moha et al. 2010) and machine learning (Fontana et al. 2015; Maiga et al. 2012; Kreimer 2005). In these settings, rules are manually or automatically 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 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. In addition, the translation of symptoms into rules is not obvious due to the ambiguities related to the definition of some code-smells. In fact, the same symptom could be associated with many code-smell types, which may compromise the precise identification of code-smell types.

To address the above-mentioned limitations, we propose in this paper a multi-objective search-based approach for the generation of code-smell detection rules from code-smell and well-designed examples. The process aims at finding the combination of software metrics, from an exhaustive list of metric combinations, that: (1) maximizes the coverage of a set of code-smell examples collected from different systems and (2) minimizes the detection of examples of good design practices. In fact, it is difficult to ensure that the used design defect examples cover all possible bad design practices. Thus, we used good design practices as another objective to detect code fragments that are not similar to the well-designed code examples and design defect examples. In fact, several defects cannot be detected using only the bad design examples due to the lack of coverage of all possible examples of bad design practices. Thus, a deviation of well-designed code examples can be also an indicator of a design defect. In other words, well-designed code helps in minimizing the false positives in the generated detection rules. To this end, a multi-objective genetic programming (MOGP) (Deb 2001; Langdon et al. 2008) is used to generate the code-smell detection rules that find trade-offs between the two above-mentioned objectives. MOGP is a powerful evolutionary metaheuristic which extends the generic model of learning to the space of programs. The primary contributions of this paper can be summarized as follows:

- The paper introduces a novel formulation of the code-smell detection problem as a multi-objective problem that takes into account both good and bad design examples to generate detection rules. To the best of our knowledge, and based on recent surveys (Harman et al. 2012; Rasool and Arshad 2015; Abbes et al. 2011; Hall et al. 2014), this

is the first work to use multi-objective evolutionary algorithms for code-smell detection.
- The paper gives an evaluation of our multi-objective approach on seven open-source systems using existing benchmarks (Kessentini et al. 2010, 2011b; Khomh et al. 2009; Moha et al. 2010) to detect five different types of code-smells. We report the statistical analysis of MOGP results on the efficiency and effectiveness of our approach over 51 runs (Arcuri and Briand 2011). We compared our approach to random search, a multi-objective artificial immune system (MOAIS) (Gong et al. 2008), and two existing code-smell approaches (Kessentini et al. 2010, 2011a). Our results indicate that our proposal has great promise and outperforms two existing studies (Kessentini et al. 2010, 2011a) by detecting most of the expected code-smells with an average of 87 % of precision and 92 % of recall.

The remainder of this paper is structured as follows. Section 2 provides the background required to understand our approach and the nature of the code-smell detection challenge. In Sect. 3, we describe multi-objective optimization and explain how we formulate code-smell detection as an optimization problem. Section 4 presents and discusses the results obtained by applying our approach to seven large open-source projects. Related work is discussed in Sect. 5, while in Sect. 6 we conclude and suggest future research directions.

2 Background and problem statement

Code-smells classify shortcomings in software that can decrease software maintainability. They are also defined as structural characteristics of software that may indicate a code or design problem that makes software hard to evolve and maintain, and trigger refactoring of code (Yamashita and Moonen 2012; Brown et al. 1998). Code-smells are not limited to design flaws since most of them occur in code and are not related to the original design. In fact, most of code-smells represent patterns or aspects of software design that may cause problems in the further development and maintenance of the system (Tufano et al. 2015). As stated by Fowler et al. (1999), code-smells are unlikely to cause failures directly, but may do it indirectly. In general, they make a system difficult to change, which may in turn introduce bugs. Different types of code-smells, presenting a variety of symptoms, have been studied in the intent of facilitating their detection and suggesting improvement solutions. A list of code-smells is defined in the literature with their potential symptoms (Yamashita and Moonen 2012, 2013; Sjøberg et al. 2013; Mäntylä 2010).

In our approach, we focus on the following *five* code-smell types:

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

We choose these code-smell types in our experiments because they are the most frequent to detect and fix based on recent empirical studies (Maiga et al. 2012; Al Dallal 2015; Palomba et al. 2014; Sahin et al. 2014). Of course, the proposed approach in this paper can be extended to other types of code-smell.

Software metrics can be used to capture the structural and semantic attributes of the software, and can be a reliable indicator of the quality of design (Maiga et al. 2012; Rasool and Arshad 2015; Hall et al. 2014; Bavota et al. 2015). These quality indicators can then be used to quantitatively estimate and reflect the design signatures of software architecture in terms of many metrics including coupling, cohesion, and cyclic complexity. The code-smell detection process usually involves finding the fragments of code which violate these software metrics. Several studies (Chidamber and Kemerer 1994; Abreu et al. 1995; Fenton and Pfleeger 1998) classified these software metrics for object-oriented architectures. We selected and used in our experiments the software metrics described in Table 1.

The manual definition of rules to identify maybe difficult and can be time-consuming for some types of code-smell such as the functional decomposition defect. One of the main issues 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. Thus, the manual definition of detection rules sometimes requires a high calibration effort. Furthermore, the manual selection of the best combination of metrics that formalize some symptoms of code-smells is challenging. In addition, the translation of code-smell definitions into metrics is not straightforward. Some definitions of code-smells are confusing, and it is difficult to determine which metrics to use to identify such design problems. To address these challenges, we describe in the next section our approach based on the use of multi-objective genetic programming to generated code-smell detection rules using not only bad design practice examples but also good ones.

3 Code-smell detection as a multi-objective problem

In this section, we describe first the principle of multi-objective genetic programming (MOGP), and then we give details about our adaptation of MOGP to the problem of code-smell detection.

3.1 Multi-objective genetic programming: an overview

Genetic programming (GP) is a powerful evolutionary metaheuristic which extends the generic model of learning to the space of programs (Langdon et al. 2008). Differently to other evolutionary approaches, in GP, population individuals are themselves programs following a tree-like structure instead of fixed length linear string formed from a limited alphabet of symbols. GP can be seen as a process of program induction that allows automatically generating programs that solve a given task. Most exiting work on GP makes use of a single-objective formulation of the optimization problem to solve using only one fitness function to evaluate the solution. Differently to single-objective optimization problems, the resolution of multi-objective optimization problems (MOPs) yields a set of trade-off solutions called non-dominated solutions, and their image in the objective space is called the Pareto front. In what follows, we give some background definitions related to this topic:

Definition 1 (*MOP*) A MOP consists in minimizing or maximizing a set of objective functions under some constraints (Deb 2001). An MOP could be expressed as:

$$\begin{cases} \text{Min}\, f(x) = [f_1(x), f_2(x), \ldots, f_M(x)]^T & \\ g_j(x) \geq 0 & j = 1, \ldots, P; \\ h_k(x) = 0 & k = 1, \ldots, Q; \\ x_i^L \leq x_i \leq x_i^U & i = 1, \ldots, n. \end{cases} \quad (1)$$

where M is the number of objective functions, P is the number of inequality constraints, Q is the number of equality constraints, x_i^L and x_i^U correspond to the lower and upper bounds of the variable x_i. A solution x_i satisfying the $(P + Q)$ constraints is said feasible and the set of all feasible solutions defines the feasible search space denoted by Ω. In this formulation, we consider a minimization MOP since maximization can be easily turned to minimization based on the duality principle by multiplying each objective function by -1. The resolution of a MOP consists in approximating the whole Pareto front (Aghezzaf and Hachimi 2000).

Definition 2 (*Pareto optimality*) A solution $x^* \in \Omega$ is Pareto optimal if there does not exist any solution x such that $f_m(x) < f_m(x^*)$ for all m.

The definition of Pareto optimality states that x^* is Pareto optimal if no feasible vector x exists which would improve some objective without causing a simultaneous worsening in at least another one. Other important definitions associated with Pareto optimality are essentially the following:

Definition 3 (*Pareto dominance*) A solution $u = (u_1, u_2, \ldots, u_n)$ is said to dominate another solution $v = (v_1, v_2, \ldots, v_n)$ (denoted by $f(u) \preceq f(v)$) if and only if $f(u)$ is partially less than $f(v)$. In other words, $\forall m \in \{1, \ldots, M\}$ we have $f_m(u) \leq f_m(v)$ and $\exists m \in \{1, \ldots, M\}$ where $f_m(u) < f_m(v)$.

Definition 4 (*Pareto optimal set*) For a MOP $f(x)$, the Pareto optimal set is $P^* = \{x \in \Omega | \neg \exists x' \in \Omega, f(x') \preceq f(x)\}$.

Definition 5 (*Pareto optimal front*) For a given MOP $f(x)$ and its Pareto optimal set P^*, the Pareto front is $\text{PF}^* = \{f(x), x \in P^*\}$.

In this work, we use a MOGP that follows the same evolutionary process of NSGA-II (Deb et al. 2002). As described in Fig. 1, the first step in MOGP is to create randomly a population P_0 of individuals encoded as trees. Then, a child population Q_0 is generated from the population of parents P_0 using genetic operators such as crossover and mutation. The crossover operator is based on sub-trees exchange, and the mutation is based on random change in the tree. Both populations are merged into an initial population R_0 of size N, and a subset of individuals is selected, based on the dominance principle and crowding distance (Deb et al. 2002) to create the next generation. This process will be repeated until reaching the last iteration according to stop criteria. In the section, we describe the adaptation of MOGP to our problem.

Table 1 List of software metrics

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

3.2 MOGP adaptation for code-smell detection

3.2.1 Problem formulation

The code-smell detection problem involves searching for the best metric combinations among the set of candidate ones, which constitutes a huge search space. A solution of our code-smell detection problem is a set of rules (metric combination with their thresholds values) where the goal of applying these rules is to detect code-smells in a system. We propose a multi-objective formulation of the code-smell rule generation problem. Consequently, we have two objective functions to be optimized: (1) maximizing the coverage of code-smell examples and (2) minimizing the detection of good design practice examples.

Fig. 1 High-level pseudo-code of MOGP

```
01. Begin
02. While stopping criteria not reached do
03.   $R_t \leftarrow P_t \cup Q_t$;
04.   $F \leftarrow$ fast-non-dominated-sort ($R_t$) ;
05.   $P_{t+1} \leftarrow \emptyset$ ; i ← 1 ;
06    While | $P_{t+1}$| + |$F_i$| ≤ $N$ do
07.     Apply crowding-distance-assignment ($F_i$) ;
08.     $P_{t+1} \leftarrow P_{t+1} \cup F_i$;
09      $i \leftarrow i + 1$ ;
10.   End While
11.   Sort (Fi, $\prec n$ ) ;
12.   $P_{t+1} \leftarrow P_{t+1} \cup F_i[1 : N - |P_{t+1}|]$ ;
13.   $Q_{t+1} \leftarrow$ create-new-population ($P_{t+1}$) ;
14.   $t \leftarrow t+1$ ;
15. End While
16. End
```

The collected examples of well-designed code and code-smells on different systems are taken as an input for our approach. Analytically speaking, the formulation of the multi-objective problem can be stated as follows:

$$\begin{cases} \max f_1(x) = \dfrac{\frac{|\mathrm{DCS}(x)| \cap |\mathrm{ECS}|}{|\mathrm{ECS}|} + \frac{|\mathrm{DCS}(x)| \cap |\mathrm{ECS}|}{|\mathrm{DCS}(x)|}}{2} \\ \min f_2(x) = \dfrac{\frac{|\mathrm{DCS}(x)| \cap |\mathrm{EGE}|}{\mathrm{EGE}} + \frac{|\mathrm{DCS}(x)| \cap |\mathrm{EGE}|}{|\mathrm{DCS}(x)|}}{2} \end{cases} \tag{2}$$

where |DCS(x)| is the cardinality of the set of detected code-smells by the metric combination x, |ECS| is the cardinality of the set of existing code-smells, and |GDE| is the cardinality of the set of existing good examples.

Once the bi-objective trade-off front is obtained, the developer can navigate through this front in order to select his/her preferred solution (metric combination).

3.2.2 Solution approach

We use a MOGP that follows the same evolutionary scheme as the well-known multi-objective evolutionary algorithm NSGA-II to try to solve the code-smell detection problem. As noted by Harman et al. (2012), a multi-objective algorithm cannot be used "out of the box"—it is necessary to define problem-specific genetic operators to obtain the best performance. To adapt MOGP to our problem, the required steps are to create: (1) solution representation, (2) solution variation, and (3) solution evaluation.

3.2.2.1 Solution representation In our MOGP, a solution is composed of terminals and functions. Therefore, when applying MOGP to solve a specific problem, they should be carefully selected and designed to satisfy the requirements of the current problem. The

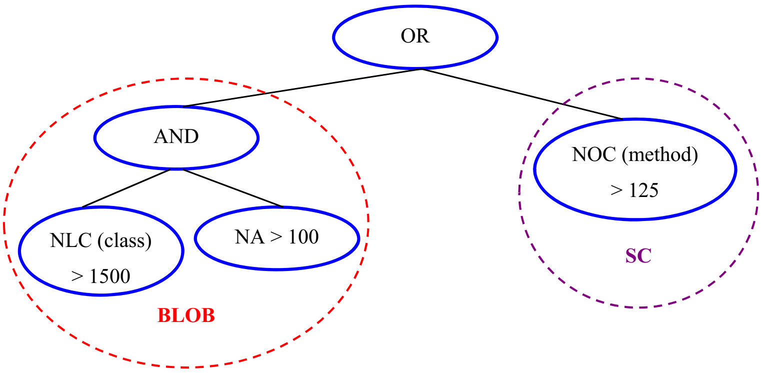

Fig. 2 Example of a detection rule according to our formulation

terminals correspond to different software metrics with their threshold values. The functions that can be used between these metrics are union (OR) and intersection (AND). More formally, each candidate solution x in this problem is a set of detection rules where each rule is represented by a binary tree such that:

1. Each leaf node $(\text{Terminal})_L$ belongs to the set of metrics (such as number of methods, number of attributes) discussed in Sect. 2, and their corresponding thresholds are generated randomly.
2. Each internal node (Functions) N belongs to the connective (logic operators) set $C = \{\text{AND, OR}\}$.

The set of candidates solutions (rules) corresponds to a logic program that is represented as a forest of AND–OR trees. Each sub-tree corresponds to a rule for the detection of specific code-smell (e.g., blob, functional decomposition). Figure 2 illustrates an example of a solution according to our formulation including two rules. It is an example of rules generated randomly by the genetic programming and not the best set of rules found at the end of the execution of the algorithm. The first rule is to detect blob using the two metrics NLC and NA and the second rule detects spaghetti code (SC) using one metric NOC. The thresholds value is selected randomly along with the comparison and logic operators.

3.2.2.2 Solution variation The mutation operator can be applied to a function node or a terminal node. It starts by randomly selecting a node in the tree. Then, if the selected node is a terminal (quality metric), it is replaced by another terminal (metric or another threshold value). If it is a function (AND–OR), it is replaced by a new function. If a tree mutation is to be carried out, the node and its sub-tree are replaced by a new randomly generated sub-tree. Figure 3 illustrates an example of a mutation operation. One node is deleted from the tree representation in Fig. 3 to generate a new other possible solution. For the crossover, two parent individuals are selected and a sub-tree is picked on each one. Then crossover swaps the nodes and their relative sub-trees from one parent to the other. This operator must ensure the respect of the depth limits. The crossover operator can be applied with only parents having the same rules category (code-smell type to detect). Each child thus obtains information from both parents. Figure 4 describes an example of a crossover operation. Two sub-trees are exchanges between two solutions to generate two new ones.

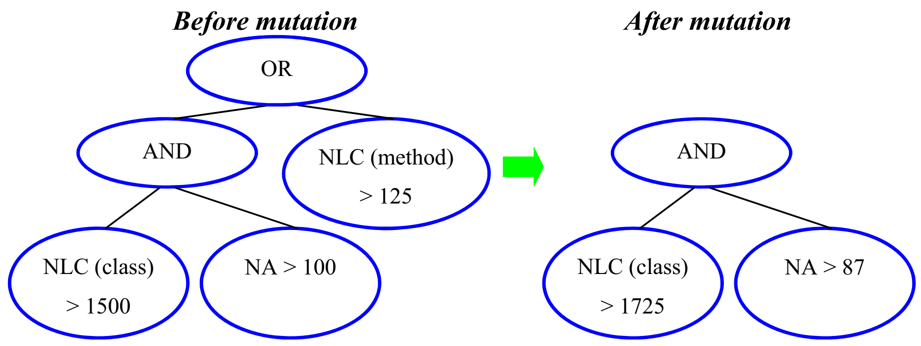

Fig. 3 Exemplified mutation operation

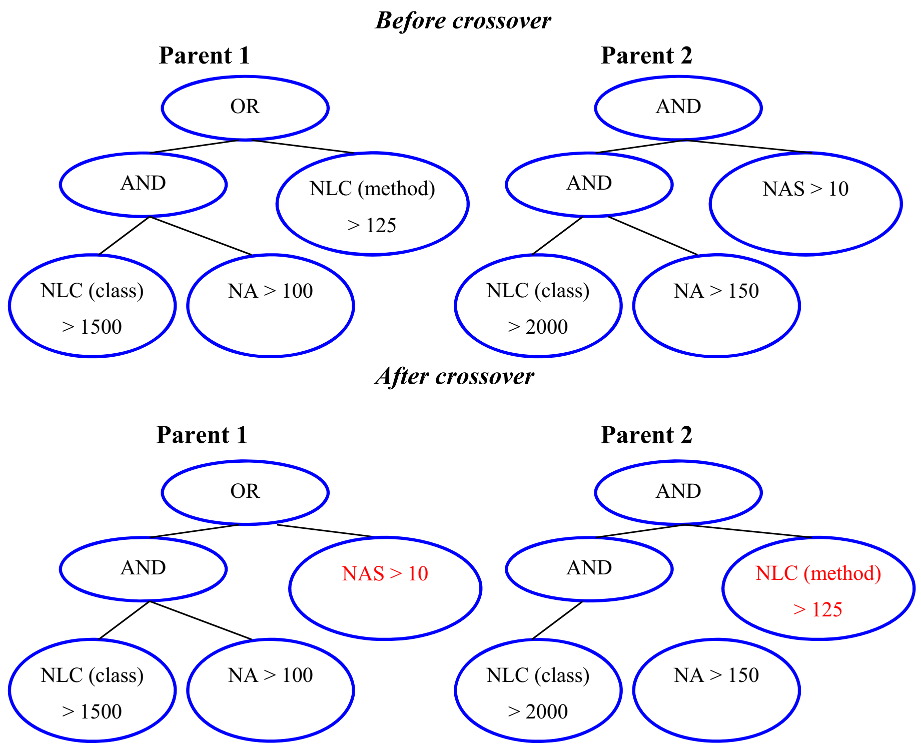

Fig. 4 Exemplified corssover operation

3.2.2.3 Solution evaluation The solution is evaluated based on the two objective functions defined in the previous section. Since we are considering a bi-objective formulation, we use the concept of Pareto optimality to find a set of compromise (Pareto optimal) solutions. The fitness of a particular solution in MOGP corresponds to a couple (*Pareto Rank*, *Crowding distance*). In fact, MOGP classifies the population individuals (of parents and children) into different layers, called non-dominated fronts. Non-dominated solutions are assigned a rank of 1 and then are discarded temporary from the population. Non-dominated solutions from the truncated population are assigned a rank of 2 and then are discarded temporarily. This process is repeated until the entire population is classified with the domination metric. After that, a diversity measure, called *crowding distance*, is assigned front-wise to each individual. The crowding distance is the average side length of the cuboid formed by the nearest neighbors of the considered solution. Once each solution is assigned its Pareto rank, mating selection and environmental selection are performed. This is based on the crowded comparison operator ($\prec n$) that favors solutions having better Pareto ranks, and in case of equal ranks, it favors the solution having larger crowding distance. In this way, convergence toward the Pareto optimal bi-objective front and diversity along this front are emphasized simultaneously. The output of MOGP is the last obtained parent population containing the best of the non-dominated solutions found. When plotted in the objective space, they form the Pareto front from which the user will select his/her preferred code-smell detection rules solution.

4 Empirical evaluation

4.1 Design of the experimental study

4.1.1 Research questions

We defined five research questions that address the applicability, performance in comparison with existing refactoring approaches, and the usefulness of our multi-objective code-smell detection approach. The five research questions are as follows:

RQ1 How does MOGP perform against random search (this serves as sanity check)? If random search outperforms an intelligent search method, then we can conclude that our problem formulation is not adequate.

RQ2 How does MOGP perform against start-of-the-art code-smell detectors?

RQ2.1 How does MOGP perform compared to MOAIS? It is important to justify the use of MOGP for the problem of multi-objective code-smell detection. It is almost impossible to formally justify the use of a specific metaheuristic search against another. The only proof in general is the experiments' result. To this end, we compared MOGP with another multi-objective algorithm, MOAIS (Gong et al. 2008) using the same adaptations. MOAIS is a widely used multi-objective non-dominated neighbor-based selection algorithm, which is used as a benchmark for evaluation of multi-object search algorithms.

RQ2.2 How does MOGP perform compared to mono-objective genetic programming (GP) with aggregation of both objectives? This research question is essential to validate the choice of using multi-objective algorithm as opposed to using mono-objective algorithm.

RQ2.3 How does MOGP perform compared to some existing code-smell detection approaches? It is important to determine whether the proposed detection approach which employs good and bad design examples performs better than existing approaches including non-search techniques (Kessentini et al. 2010, 2011a).

RQ3 To what extent can our approach generate rules that detect different code-smell types? It is important to discuss the ability of our approach to detect different types of code-smells to evaluate the quality of each detection rule separately.

Table 2 Software projects features

Systems	Number of classes	Number of code-smells				
		Blob	FD	SC	DC	FE
ArgoUML v0.26	1358	22	52	64	21	14
ArgoUML v0.3	1409	10	58	61	16	22
Xerces v2.7	991	14	32	36	19	24
Ant-Apache v1.5	1024	22	47	34	23	21
Ant-Apache v1.7.0	1839	25	51	48	18	26
Gantt v1.10.2	245	4	21	16	14	11
Azureus v2.3.0.6	1449	19	44	52	29	34

4.1.2 Studied systems

Our study considers the extensive evolution of different open-source Java systems analyzed in the literature (Kessentini et al. 2010, 2011a). As described in Table 2, the corpus used includes releases of Apache Ant[1], ArgoUML[2], Gantt[3], Azureus[4] and Xerces-J[5]. Apache Ant is a build tool and library specifically conceived for Java applications. ArgoUML is an open-source UML modeling tool. Xerces is a family of software packages that implement a number of standard APIs for XML parsing. GanttProject is a tool for creating project schedules in the form of Gantt charts and resource-load charts. Azureus is a peer-to-peer file-sharing tool. Table 2 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 in total for the five types of code-smell considered in our experiments. Indeed, in several works (Kessentini et al. 2010, 2011a; Khomh et al. 2009; Palomba et al. 2013), the authors asked different groups of developers to analyze the libraries to tag instances of specific code-smells to validate their detection techniques. For replication of experimental results, they provided a corpus describing instances of different code-smells including blob, spaghetti code, and functional decomposition. Subjects included eight master students in software engineering, five Ph.D. students in software engineering and two faculty members in software engineering. All the 15 subjects were familiar with Java development and software maintenance activities including code-smell detection and refactoring. The experience of these subjects on Java programming ranged from 1 to 17 years. In our study, we verified the capacity of our approach to detect classes that correspond to instances of these code-smells. We choose the above-mentioned open-source systems because they were analyzed in related work and for comparison purposes. JHotDraw[6] was chosen as an example of reference code (training examples for our MOGP

[1] http://ant.apache.org/.

[2] http://argouml.tigris.org/.

[3] www.ganttproject.biz.

[4] http://vuze.com.

[5] http://xerces.apache.org/xerces-j/.

[6] http://www.jhotdraw.org/.

of good design practices) because it contains very few known code-smells (Al Dallal 2014; Kessentini et al. 2010).

4.1.3 Evaluation metrics

We use the two following performance indicators (which are among the most used in multi-objective optimization) when comparing MOGP and MOAIS:[7]

- *Hypervolume* (*IHV*) (Arcuri and Briand 2011) It corresponds to the proportion of the objective space that is dominated by the Pareto front approximation returned by the algorithm and delimited by a reference point. Larger values for this metric mean better performance.
- *Inverted Generational Distance* (*IGD*) (Arcuri and Briand 2011) It is a convergence measure that corresponds to the average Euclidean distance between the Pareto front approximation *PA* provided by the algorithm and the reference front *RF* (RF is the set of non-dominated solutions obtained over all runs). The distance between *PA* and *RF* in an *M*-objective space is calculated as the average *M*-dimensional Euclidean distance between each solution in *PA* and its nearest neighbor in *RF*. Lower values for this indicator mean better performance (convergence).

To assess the accuracy of our approach, we compute two measures: (1) precision and (2) recall, originally stemming from the area of information retrieval:

- *Precision (PR)* It denotes the fraction of correctly detected code-smells among the set of all detected code-smells. It could be seen as the probability that a detected code-smell is correct.

$$\text{precision} = \frac{\{(\text{relevant code smells}) \cap (\text{detected code smells})\}}{(\text{detected code smells})}$$

- *Recall (RE)* It corresponds to 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). It could be seen as the probability that an expected code-smell is detected.

$$\text{recall} = \frac{\{(\text{relevant code smells}) \cap (\text{detected code smells})\}}{(\text{relevant code smells})}$$

4.1.4 Used inferential statistical methodology

Since metaheuristic algorithms are stochastic optimizers, they usually provide different results for the same problem instance from one run to another. For this reason, 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 (Palomba et al. 2013) with a 95 % confidence level ($\alpha = 5\ \%$). This statistical test verifies the null hypothesis H_0 that the obtained results of two algorithms are samples from continuous distributions with equal medians, as against the alternative that

[7] http://staff.unak.is/andy/StaticAnalysis0809/metrics/overview.html.

they are not, H_1. The p value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H_0, while it is true (type I error). A p value that is less than or equal to α ($\leq$0.05) means that we accept H_1 and we reject H_0. However, a p value that is strictly greater than α (>0.05) means the opposite.

4.1.5 Parameter tuning and setting

Parameter setting has a significant influence on the performance of a search algorithm on a particular problem instance. For this reason, for each multi-objective algorithm and for each system (cf. Table 3), we perform a set of experiments using several population sizes: 50, 100, 200, 500 and 1000. The stopping criterion was set to 250,000 fitness evaluations for all algorithms in order to ensure fairness of comparison. The other parameters' values were fixed by trial and error and are as follows: (1) crossover probability = 0.8; mutation probability = 0.20 where the probability of gene modification is 0.3; stopping criterion = 250,000 fitness evaluations. For MOAIS, the maximum size of the dominant population, and the clone population size are set to 100 and 20, respectively.

4.2 Results analysis

4.2.1 Results for RQ1

We do not dwell long in answering the first research question (RQ1) that involves comparing our approach based on NSGA-II with random search. The remaining research questions will reveal more about the performance, insight, and usefulness of our approach. Table 4 confirms that MOGP and MOAIS are better than random search based on the two quality indicators IHV and IGD on all seven open-source systems. The Wilcoxon rank-sum test showed that in 51 runs both MOGP and MOAIS results were significantly better than random search. We conclude that there is empirical evidence that our multi-objective formulation surpasses the performance of random search thus our formulation is adequate (this answers RQ1).

4.2.2 Results for RQ2

In this section, we compare our MOGP adaptation to the current, state-of-the-art code-smell detection approaches. To answer the second research question, RQ2.1, we compared MOGP to another multi-objective algorithm, MOAIS, using the same adaptations. Table 5 shows the overview of the results of the significance tests comparison between MOGP and MOAIS. MOGP outperforms MOAIS in most of the cases: 11 out of 14 experiments (78 %).

A more qualitative evaluation is presented in Figs. 5 and 6 illustrating the boxplots obtained for the multi-objective metrics on the different projects. We see that for almost all problems the distributions of the metrics values for MOGP have smaller variability than for MOAIS. This fact confirms the effectiveness of MOGP over MOAIS in finding a well-converged and well-diversified set of Pareto optimal detection rules solutions (RQ2.1).

Next, we use precision and recall measures to compare the efficiency of our MOGP approach compared to mono-objective GP (aggregating both objectives) and two existing code-smell detection studies (Kessentini et al. 2010, 2011a). We first note that the mono-objective approaches provide only one detection solution (set of detection rules), while

Table 3 Best population size configurations

System	MOGP	MOAIS	Mono-GP
ArgoUML v0.26	500	500	500
ArgoUML v0.3	500	200	500
Xerces v2.7	200	200	100
Ant-Apache v1.5	200	500	500
Ant-Apache v1.7.0	100	100	200
Gantt v1.10.2	100	100	100
Azureus v2.3.0.6	500	500	500

Table 4 Significantly best algorithm among random search, MOGP and MOAIS (no stat. diff. means that MOGP and MOAIS are significantly better than random, but not statistically different)

Project	IHV	IGD
ArgoUML v0.26	MOGP	MOGP
ArgoUML v0.3	MOGP	MOGP
Xerces v2.7	No stat. diff.	MOAIS
Ant-Apache v1.5	MOGP	MOGP
Ant-Apache v1.7.0	MOGP	MOGP
Gantt v1.10.2	MOGP	No stat. diff.
Azureus v2.3.0.6	MOGP	MOGP

MOGP generate a set of non-dominated solutions. In order to make meaningful comparisons, we select the best solution for MOGP using a *knee point* strategy as described in Fig. 7. The knee point corresponds to the solution with the maximal trade-off between maximizing the coverage of code-smells and minimizing the number of detected well-designed code. Thus, for MOGP, we select the knee point from the Pareto approximation having the median IHV value. We aim by this strategy to ensure fairness when making comparisons against the mono-objective EA. For the latter, we use the best solution corresponding to the median observation on 51 runs.

The results from 51 runs are depicted in Table 5. It can be seen that MOGP provides better precision and recall scores for the detection of code-smells. For recall (RE), MOGP is better than GP in 100 % of the cases. We have the same observation for the precision also where MOGP outperforms GP in all cases with an average of more than 90 %. Thus, it is clear that a multi-objective formulation of our problem outperforms the aggregation-based approach. In conclusion, we answer RQ2.2 by concluding that the results obtained in Table 5 confirm that both multi-objective formulations are adequate and outperform the mono-objective algorithm based on an aggregation of two objectives related to use of good and bad software design examples. Table 5 also shows the results of comparing our multi-objective approach based on MOGP with two mono-objective refactoring approaches (Kessentini et al. 2010, 2011a). In Kessentini et al. (2011a), the authors used search-based techniques to detect code-smells from only code-smell examples. In Kessentini et al. (2010), an artificial immune system approach is proposed to detect code-smells by deviation with well-designed code examples. It is apparent from Table 5 that MOGP outperforms both mono-objective approaches in terms of precision and recall in most of the cases. This is can be explained by the fact that our proposal takes into account both positive and negative examples when generating the detection rules. If only code-smell examples are used, then it is difficult to ensure the coverage of all possible bad design behaviors. The

Table 5 Recall and precision median values of MOGP, MOAIS, GP, Kessentini et al. (2010, 2011b) over 51 independent simulation runs

System	RE-MOGP	RE-MOAIS	RE-GP	RE (Kessentini et al. 2011a)	RE (Kessentini et al. 2010)	PR-MOGP	PR-MOAIS	PR-GP	PR (Kessentini et al. 2011a)	PR (Kessentini et al. 2010)
ArgoUML v0.26	92 % (158/173)	90 % (155/173)	85 % (147/173)	81 % (141/173)	83 % (143/173)	87 % (158/176)	85 % (155/178)	76 % (147/188)	74 % (141/194)	72 % (141/198)
ArgoUML v0.3	90 % (149/167)	88 % (146/167)	81 % (136/167)	74 % (123/167)	76 % (126/167)	86 % (149/174)	86 % (146/174)	73 % (136/179)	64 % (123/194)	63 % (123/196)
Xerces v2.7	90 % (113/125)	90 % (113/125)	83 % (103/125)	75 % (93/125)	76 % (95/125)	87 % (113/131)	85 % (113/136)	71 % (103/143)	69 % (93/132)	67 % (93/136)
Ant-Apache v1.5	89 % (131/147)	87 % (127/147)	79 % (116/147)	85 % (124/147)	82 % (120/147)	90 % (131/145)	90 % (127/145)	76 % (116/152)	75 % (124/160)	72 % (124/167)
Ant-Apache v1.7.0	93 % (157/168)	93 % (157/168)	77 % (129/168)	71 % (119/168)	69 % (115/168)	95 % (157/163)	92 % (157/169)	72 % (129/176)	67 % (119/176)	65 % (119/181)
Gantt v1.10.2	90 % (57/66)	86 % (55/66)	83 % (56/66)	77 % (51/66)	72 % (48/66)	79 % (57/73)	76 % (55/76)	63 % (56/88)	58 % (51/89)	63 % (51/82)
Azureus v2.3.0.6	94 % (169/178)	92 % (163/178)	79 % (140/178)	69 % (122/178)	72 % (128/178)	86 % (169/187)	86 % (163/187)	76 % (140/184)	67 % (122/188)	69 % (122/182)

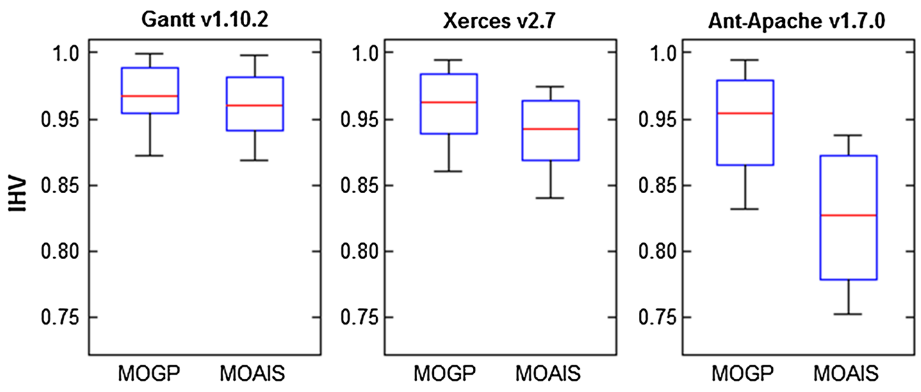

Fig. 5 IHV boxplots on three projects having different sizes (Gantt v1.10.2: small, Xerces v2.7: medium, Ant-Apache v1.7.0: large)

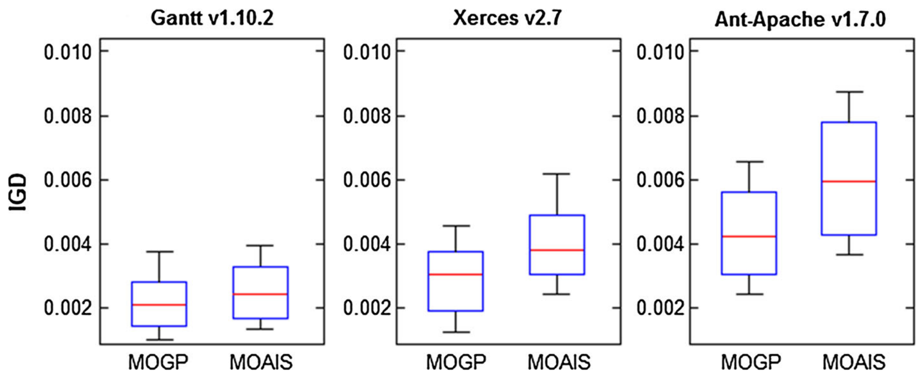

Fig. 6 IGD boxplots on three projects having different sizes (Gantt v1.10.2: small, Xerces v2.7: medium, Ant-Apache v1.7.0: large)

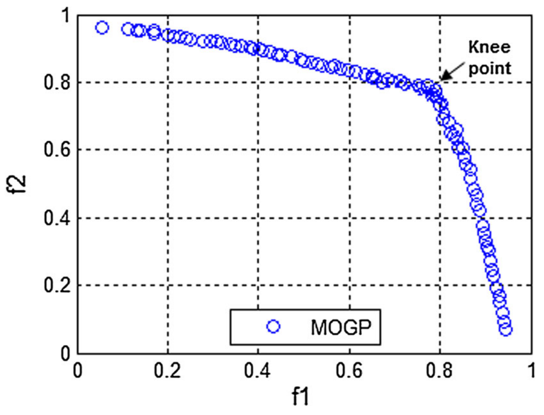

Fig. 7 Median values of precision and recall on 51 runs for the five types of code-smell

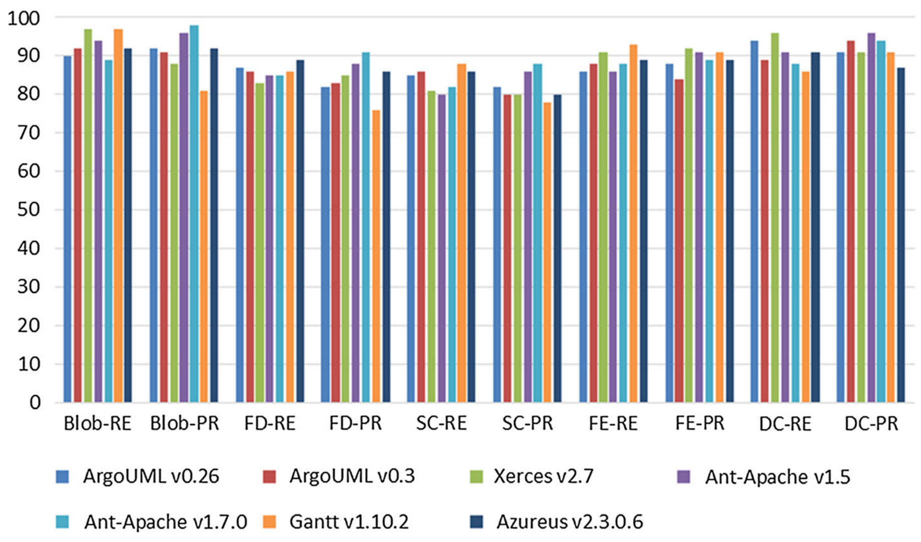

Fig. 8 MOGP knee point for bi-objective code-smell detection problem. f1 and f2 correspond to the normalized values of both fitness functions

same observation is still valid for the use only of well-designed code examples. The use of both types of examples represents a complementary way to formulate the problem of code-smell detection using a multi-objective approach. To answer RQ2.3, the results of Table 5 support the claim that our MOGP formulation outperforms, on average, the two code-smell existing approaches.

4.2.3 *Results for RQ3*

We noticed that our technique does not have a bias toward the detection of specific code-smell types. As described in Fig. 8, in all systems, we had an almost equal distribution of each code-smell types (SCs, blobs, FEs, DCs and FDs). Overall, all the five code-smell types are detected with good precision and recall scores in the different systems (more than 85 %). 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 MOGP approach detects well all the five types of considered code-smells (RQ3).

4.3 Threats to validity

In our experiments, a 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. Another construct threat is the base of good and bad design examples that we considered on our experiments that need to include a high number of examples. We can conclude that the use of open-source systems can be a good starting point to use our approach in industrial settings.

We take into consideration 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 with a 95 % 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. Another important potential hazard to results is the over reliance on boxplots while comparing different detection algorithms. Underlying distributions can cause spread in values which might not exhibit real performance of an algorithm. Therefore, it is important to run multiple instances of the experiment for result evaluation.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on seven different widely used open-source systems belonging to different domains and with different sizes, as described in Table 2. However, we cannot assert that our results can be generalized to industrial applications, other programming languages, and to other practitioners. Future replications of this study are necessary to confirm the generalizability of our findings.

5 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.

Fowler et al. (1999) 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. (1999) 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 Salehie et al. (2006), 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 high rate of false positives generated by the above-mentioned approaches encouraged other teams to explore semiautomated 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. (2004) present a pattern-based framework for developing tool support to detect software anomalies by representing potential code-smells with different colors. Dhambri et al. (2008) 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 (Langelier et al. 2005) are based on manual and human inspection, they still, not only, slow and time-consuming, but also subjective.

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

Moha et al. (2010) 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. 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 a reduced rate of false positives. Indeed, this approach has been evaluated on only four well-known design code-smells: the blob, functional decomposition, spaghetti code, and Swiss-army knife because the literature provides obvious symptom descriptions on these code-smells. Recently, another probabilistic approach has been proposed by Khomh et al. (2009) and Abbes et al. (2011) extending the DECOR approach (Moha et al. 2010), 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. Similarly, Munro et al. (2005) have proposed description and symptom-based approach using a precise definition of bad smells from the informal descriptions given by the originators Fowler and Beck. 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 (Yamashita and Moonen 2013; Hall et al. 2014). This template consists of three main parts: a code-smell name, a text-based description of its characteristics, and heuristics for its detection. Recently, several studies proposed to identify code-smells using machine learning (Fontana et al. 2015; Maiga et al. 2012; Kreimer 2005; Vidal et al. 2014). In these approaches, the classes are classified as code-smells or not using a set of training data of code-smells. The code-smells can be detected using association rules thus the output is different than our technique where the structure of the rules allows the support of logic operators. Tufano et al. (2015)

studied how code-smells are generated in the code and when programmers decided to fix them by mining several releases of open-source systems. However, they did not present a technique to generalize the results by generating detection rules for code-smells.

In another category of work, Marinescu (2004) has used software metrics to propose a mechanism called "detection strategy" for formulating metric-based rules that capture deviations from good design principles and heuristics. Detection strategies allows for locating classes or methods affected by a particular design code-smell. As such, Marinescu has defined detection strategies for capturing around ten important flaws of object-oriented design found in the literature. After his suitable symptom-based characterization of design code-smells, Salehie et al. (2006) proposed a metric-based heuristic framework to detect and locate object-oriented design flaws similar to those illustrated by Marinescu (2004). 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.

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 is no consensus in defining design smells, different threshold values should be tested to find the best ones.

Recently, several studies proposed automated techniques for the detection of "micro-patterns" (Gil and Maman 2005; Destefanis et al. 2012; Maggioni and Arcelli 2010; Concas et al. 2013). Micro-patterns are similar to design patterns, but closer to the implementation level. Micro-patterns are used to describe good programming practices and can be expressed as a formal condition on the structure of a class. Gil et al. (2005) presents a catalog of 27 micro-patterns defined on JAVA classes and interfaces to capture a wide spectrum of common programming practices, including the use of inheritance, data management and wrapping. Destefanis et al. (2012) showed empirically that there is high correlation between the classes that did not include micro-patterns and faults. Maggioni et al. (2010) proposed a metric-based approach to identify micro-patterns and Concas et al. (2013) found that the distribution of micro-patterns in software systems developed using Agile methodologies does not differ from the distribution studied in other systems. An interesting future research direction can be to study the correlation between anti micro-patterns and code-smells on different software systems.

Our approach is inspired by contributions in the domain of search-based software engineering (SBSE) (Harman et al. 2012). 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. Based on recent SBSE surveys (Harman et al. 2012), the use of multi-objective optimization is still limited in software engineering. Recently, Mkaouer et al. (2015) proposed a many-objective approach to recommend refactorings in order to fix code-smells. The proposed algorithm search for the best combination of refactorings that can improve six quality attributes such as maintainability, effectiveness. However, the problem of the detection of code-smells was not addressed and a list of code-smells can be taken as input to be fixed by the suggested refactorings. To the best of our knowledge, we are proposing in this paper the first approach to detect code-smells using multi-objective optimization.

6 Conclusion and future work

In this paper, we introduced a novel multi-objective approach to generate rules for the detection of code-smells. To this end, we used MOGP to find the best trade-offs between maximizing the detection of examples of code-smells and minimizing the detection of well-designed code examples. We evaluated our approach on seven large open-source systems. All results were found to be statistically significant over 51 independent runs using the Wilcoxon rank-sum test with a 99 % confidence level ($\alpha < 1$ %). Our MOGP results outperform some existing studies (Kessentini et al. 2010, 2011a) by detecting most of the expected code-smells with an average of 87 % of precision and 92 % of recall.

Future work should validate our approach with additional code-smell types in order to conclude about the general applicability of our methodology. Also, in this paper, we only focused on the detection of code-smells. We are planning to extend the approach by automating the correction of these 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 based on a severity score.

References

Abbes, M., Khomh, F., Gueheneuc, Y.-G., & Antoniol, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Software maintenance and reengineering (CSMR), 2011 15th European conference on* (pp. 181–190). IEEE.

Abreu, F., Goulão, M., & Esteves, R. (1995). Toward the design quality evaluation of object-oriented software systems. In *Proceedings of 5th ICSQ*.

Aghezzaf, B., & Hachimi, M. (2000). Generalized invexity and duality in multiobjective programming problems. *Journal of Global Optimization, 18*(1), 91–101.

Al Dallal, J. (2014). Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and Software Technology, 58*, 231–249.

Al Dallal, J. (2015). Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and Software Technology, 58*, 231–249.

Arcuri, A., & Briand, L. C. (2011). A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd international conference on software engineering (ICSE)* (pp. 1–10).

Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., & Palomba, F. (2015). An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software, 107*, 1–14.

Brown, W. J., Malveau, R. C., Brown, W. H., & Mowbray, T. J. (1998). *Anti-patterns: Refactoring software, architectures, and projects in crisis*. Hoboken: Wiley.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering, 20*(6), 293–318.

Concas, G., Destefanis, G., Marchesi, M., Ortu, M., & Tonelli, R. (2013). *Micro patterns in agile software*. Berlin: Springer.

Deb, K. (2001). *Multiobjective optimization using evolutionary algorithms*. New York: Wiley.

Deb, K., Agrawal, S., Pratap, A., & Meyarivan, T. (2002). A fast and elitist multi-objective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation, 6*(2), 182–197.

Destefanis, G., Tonelli, R., Tempero, E., Concas, G., & Marchesi, M. (2012, September). Micro pattern fault-proneness. In *Software engineering and advanced applications (SEAA), 2012 38th EUROMICRO conference on* (pp. 302–306). IEEE.

Dhambri, K., Sahraoui, H. A., & Poulin, P. (2008). Visual detection of design anomalies. In *CSMR. IEEE* (pp. 279–283).

Fenton, N., & Pfleeger, S. L. (1998). *Software metrics: A rigorous and practical approach* (2nd ed.). London: International Thomson Computer Press.

Fontana, F. A., Mäntylä, M. V., Zanoni, M., & Marino, A. (2015). Comparing and experimenting machine learning techniques for code smell detection. In *Empirical Software Engineering* (pp. 1–49).

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring—Improving the design of existing code*. Boston: Addison-Wesley Professional.

Gil, J. Y., & Maman, I. (2005). Micro patterns in Java code. In *ACM SIGPLAN Notices* (Vol. 40, no. 10). ACM.

Gong, M., Jiao, L., Du, H., & Bo, L. (2008). Multiobjective immune algorithm with nondominated neighbor-based selection. *Evolutionary Computation, 6*(2), 225–255.

Hall, T., Zhang, M., Bowes, D., & Sun, Y. (2014). Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM), 23*(4), 33.

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

Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., & Ouni, A. (2011a). Design defects detection and correction by example. In *Proceedings of the 19th IEEE international conference on program comprehension (ICPC'11)* (pp. 81–90).

Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., & Ouni, A. (2011b). Design defects detection and correction by example. In *19th IEEE international conference on program comprehension (ICPC), (22–24 June 2011), Kingston, Canada* (pp. 81–90).

Kessentini, M., Vaucher, S., & Sahraoui, H. (2010). Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In *Proceedings of the 25th IEEE/ACM international conference on automated software engineering (ASE)* (pp. 141–151).

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., & Sahraoui, H. (2009). A Bayesian approach for the detection of code and design smells. In *Proceedings of the ICQS'09.*

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

Kreimer, J. (2005). Adaptive detection of design flaws. *Electronic Notes in Theoretical Computer Science, 141*(4), 117–136.

Langdon, W. B., Poli, R., McPhee, N. F., & Koza, J. R. (2008). Genetic programming: An introduction and tutorial, with a survey of techniques and applications. In J. Fulcher & L. C. Jain (Eds.), *Computational intelligence: A compendium* (pp. 927–1028). Berlin: Springer.

Langelier, G., Sahraoui, H. A., & Poulin, P. (2005). Visualization-based analysis of quality for large-scale software systems. In T. Ellman & A. Zisma (Eds.), *Proceedings of the 20th international conference on automated software engineering*. New York: ACM Press.

Maggioni, S., & Arcelli, F. (2010). Metrics-based detection of micro patterns. In *Proceedings of the 2010 ICSE workshop on emerging trends in software metrics*. ACM.

Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Guéhéneuc, Y. G., & Aimeur, E. (2012, October). Smurf: A svm-based incremental anti-pattern detection approach. In *Reverse engineering (WCRE), 2012 19th working conference on* (pp. 466–475). IEEE.

Mäntylä, M. V. (2010). Empirical software evolvability—code smells and human evaluations. In *ICSM* (pp. 1–6).

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

Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of ICM'04* (pp. 350–359).

Mkaouer, M. W., Kessentini, M., Bechikh, S., Cinnéide, M. Ó., & Deb, K. (2015). On the use of many quality attributes for software refactoring: A many-objective search-based software engineering approach. In *Empirical Software Engineering* (pp. 1–43).

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

Munro, M. J. (2005). Product metrics for automatic identification of "Bad Smell" design problems in java source-code. In *Proceedings of the 11th international software metrics symposium.*

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., & De Lucia, A. (2014). Do they really smell bad? A study on developers' perception of bad code smells. In *Software maintenance and evolution (ICSME), 2014 IEEE international conference on* (pp. 101–110). IEEE.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., & Poshyvanyk, D. (2013). Detecting bad smells in source code using change history information. In *Automated software engineering (ASE), 2013 IEEE/ACM 28th international conference on* (pp. 268–278). IEEE.

Rasool, G., & Arshad, Z. (2015). A review of code smell mining techniques. *Journal of Software: Evolution and Process, 27*(11), 867–895.

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

Salehie, M., Li, S., & Tahvildari, L. (2006). A metric-based heuristic framework to detect object-oriented design flaws. In *Proceedings of the 14th IEEE ICPC'06*.

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

Travassos, G., Shull, F., Fredericks, M., & Basili, V. R. (1999). Detecting defects in object-oriented designs: Using reading techniques to increase software quality. In *Proceedings of the 14th conference on object-oriented programming, systems, languages, and applications* (pp. 47–56). New York: ACM Press.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., et al. (2015). When and why your code starts to smell bad. In *ICSE*.

Van Emden, V. & Moonen, L. (2002). Java quality assurance by detecting code smells. In *Proceedings of the ninth working conference on reverse engineering (WCRE'02). IEEE computer society, Washington, DC, USA* (p. 97).

Vidal, S. A., Marcos, C., & Díaz-Pace, J. A. (2014). An approach to prioritize code smells for refactoring. In *Automated Software Engineering* (pp. 1–32).

Yamashita, A. F. & Moonen, L. (2012) Do code smells reflect important maintainability aspects? In *ICSM*, pp. 306–315.

Yamashita, A. F., & Moonen, L. (2013a). To what extent can maintenance problems be predicted by code smell detection? An empirical study. *Information & Software Technology, 55*(12), 2223–2242.

Yamashita, A. F., & Moonen, L. (2013b). To what extent can maintenance problems be predicted by code smell detection? An empirical study. *Information and Software Technology, 55*(12), 2223–2242.

Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M., & da Fonseca, V. G. (2003). Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Transaction on Evolutionary Computation, 7*(2), 117–132.

Usman Mansoor is currently a Ph.D. student in computer science at the University of Michigan under the supervision of Prof. Marouane Kessentini (University of Michigan). He is a member of the SBSE@Michigan research laboratory, University of Michigan, USA. Previously he was a Graduate Research Student of Computer Engineering in Ajou University, South Korea, under Brain Korea (BK21) scholarship initiative undertaken by Korean Government. His research interests include the application of artificial intelligence techniques to software engineering (search-based software engineering), software refactoring, software quality, and model-driven engineering.

Dr. Marouane Kessentini is a tenure-track assistant professor at University of Michigan. He is the founder of the research group: Search-based Software Engineering@Michigan. He holds a Ph.D. in computer science, University of Montreal (Canada), 2011. His research interests include the application of computational search techniques to software engineering (search-based software engineering), refactoring, software testing, model-driven engineering, software quality, and re-engineering. He has published around 50 papers in conferences, workshops, books, and journals including three best paper awards. He has served as program committee/organization member in several conferences and journals. He is the general chair of SSBSE2016.

Dr. Bruce R. Maxim is Associate Professor of Computer and Information Science at the University of Michigan, Dearborn. His research interests include: software engineering, human computer interaction, game design, social media, artificial intelligence, and computer science education. He has published a number of papers on the animation of computer algorithms, game development, and educational computing applications. Prior to coming to the University of Michigan—Dearborn, Dr. Maxim was in charge of research information systems at the University of Michigan Department of Postgraduate Medicine. He has served as the instructional computing supervisor for the University of Michigan, Medical Campus Learning Resource Center in Ann Arbor. He previously worked as a statistical programmer at the University's School of Public Health and taught Mathematics, Statistics, and Computer Science at Greenhills School in Ann Arbor. He holds memberships in three academic honor societies: Sigma Xi (Research), Upsilon Pi Epsilon (Computer Science), and Pi Mu Epsilon (Mathematics). He is also a member of the Association of Computing Machinery (Senior Member 2012), IEEE Computer Society, ASEE Software Engineering Constituent Committee, and International Game Developers Association.

Dr. Kalyanmoy Deb is Koenig Endowed Chair Professor at Electrical and Computer Engineering in Michigan State University, USA. Prof. Deb's research interests are in evolutionary optimization and their application in optimization, modeling, and machine learning. Prof. Deb has numerous awards and honors in his name, including the prestigious Shanti Swarup Bhatnagar Prize in Engineering Sciences in 2005, 'Thomson Citation Laureate Award', an award given to an Indian Researcher for making most highly cited research contribution during 1996–2005 in a particular discipline according to ISI Web of Science., Friedrich Wilhelm Bessel Research Award and Humboldt Fellowship from Alexander von Humboldt Foundation, Germany. He is a fellow of Indian National Science Academy (INSA), Indian National Academy of Engineering (INAE), Indian Academy of Sciences (IASc), and International Society of Genetic and Evolutionary Computation (ISGEC). He has been awarded 'Distinguished Alumnus Award' from his Alma mater IIT Kharagpur in 2011. Author of more than 275 research papers, two textbooks, 17 edited books, his 2001 book on Evolutionary Multiobjective Optimization Algorithms is the first ever compilation of multi-objective optimization algorithms. Because of his pioneering research in the field of evolutionary multi-objective optimization (EMO), he has been invited to present 35 Keynote lectures and more than 100 invited lectures and tutorials on the topic. His NSGA-II paper from IEEE Transactions on Evolutionary Computation (2000) is judged as the Fast-Breaking Paper in Engineering by ESI Web of Science and now this paper is awarded the 'Current Classic' and 'Most Highly Cited Paper' by Thomson Reuters. He is fellow of IEEE and three science academies in India. He has published 350+ research papers with Google Scholar citation of 55,000+ with h-index 77. He is in the editorial board on 20 major international journals. More information about his research can be found from http://www.egr.msu.edu/~kdeb.