High Dimensional Search-based Software Engineering: Finding Tradeoffs among 15 Objectives for Automating Software Refactoring using NSGA-III

Wiem Mkaouer, Marouane Kessentini, Slim Bechikh
University of Michigan, USA
firstname@umich.edu

Kalyanmoy Deb
Michigan State University, USA.
kdeb@egr.msu.edu

Mel Ó Cinnéide
University College Dublin, Ireland.
mel.ocinneide@ucd.ie

COIN Report Number 2013002

ABSTRACT
There is a growing need for scalable search-based software engineering approaches that address software engineering problems where a large number of objectives are to be optimized. Software refactoring is one of these problems where a refactoring a sequence is sought that optimizes several software metrics. Most of the existing refactoring work uses a large set of quality metrics to evaluate the software design after applying refactoring operations, but current search-based software engineering approaches are limited to using a maximum of five metrics. We propose for the first time a scalable search-based software engineering approach based on a newly proposed evolutionary optimization method NSGA-III where there are 15 different objectives to be optimized. In our approach, automated refactoring solutions are evaluated using a set of 15 distinct quality metrics. We evaluated this approach on seven large open source systems and found that, on average, more than 92% of code smells were corrected. Statistical analysis of our experiments over 31 runs shows that NSGA-III performed significantly better than two other many-objective techniques (IBEA and MOEA/D), a multi-objective algorithm (NSGA-II) and two mono-objective approaches, hence demonstrating that our NSGA-III approach represents the new state of the art in fully-automated refactoring.

Categories and Subject Descriptors
D.2 [**Software Engineering**].

General Terms
Algorithms, Reliability.

Keywords
Software quality, Search-based software engineering, Refactoring.

1. INTRODUCTION
Search-based software engineering (SBSE) studies the application of meta-heuristic optimization techniques to software engineering problems [1]. Once a software engineering task is framed as a search problem, by defining it in terms of solution representation, objective function, and solution change operators, there are a multitude of search algorithms that can be applied to solve that problem. Search-based techniques are widely applied to solve software engineering problems such as in testing, modularization, refactoring, planning, etc. [20][18].

Based on recent a SBSE survey [20], the majority of existing work treats software engineering (SE) problems from a single-objective point of view, where the main goal is to maximize or minimize one objective, e.g., correctness, quality, etc. However, most SE problems are naturally complex in which many conflicting objectives need to be optimized such as model transformation, design quality improvement, test suite generation etc. The number of objectives to consider for most of software engineering problems is, in general, high (more than three objectives); such problems are termed *many-objective*. We claim that the reason that software engineering problems have not been formulated as many-objective problems is because of the challenges in constructing a many-objective solution. In this context, the use of traditional multi-objective techniques, e.g., NSGA-II [12], widely used in SBSE, is clearly not sufficient.

There is a growing need for scalable search-based software engineering approaches that address software engineering problems where a large number of objectives are to be optimized. Improving the scalability of SBSE approaches will increase their applicability in industry and real-world settings. Recent work in optimization has proposed several solution approaches to tackle many-objective optimization problems [37], [22], [6], [5], [46] using e.g., objective reduction, new preference ordering relations, decomposition, etc. However, these techniques have not yet been widely explored in SBSE [9]. To the best of our knowledge and based on recent SBSE surveys [20], only one work exists proposed by Abdel Salam et al. [38][39] that uses a many-objective approach, IBEA (Indicator-Based Evolutionary Algorithm) [47], to address the problem of software product line creation. However, the number of considered objectives is limited to five.

Software refactoring is one of those software engineering problems where there are several objectives to be satisfied. Refactoring improves the design of a system by changing its internal structure without altering its external behavior [33][17], and is widely used to fix code smells. Code smells are known to have a negative impact on quality attributes such as flexibility or maintainability [7][45]. Software engineers often introduce code smells unintentionally during the initial design or during software development due to bad design decisions, ignorance or time pressure. Most of the existing refactoring work uses a set of more than five quality metrics to evaluate the quality of software design after applying refactoring operations. In this paper, we propose for the first time a scalable search-based software engineering approach based on NSGA-III [10] where there are 15 different objectives to optimize. Thus, in our approach, automated refactoring solutions will be evaluated using a set of 15 software quality metrics. NSGA-III is a very recent many-objective algorithm proposed by Deb et al. [10]. The basic framework remains similar to the original NSGA-II algorithm [12], with significant changes in its selection mechanism. This paper represents the first real-world application of NSGA-III and the first scalable work that supports the use of 15 objectives to address a software engineering problem.

We implemented our approach and evaluated it on seven large open source systems [48][54][49][50][51] and found that, on average, more than 92% of code smells were corrected. The statistical analysis of our experiments over 31 runs shows that NSGA-III performed significantly better than two other many-objective techniques (IBEA and MOEA/D), a multi-objective algorithm (NSGA-II) and two mono-objective approaches [25][32].

The remainder of this paper is structured as follows. Section 2 provides an overview of many-objective optimization techniques and potential applications to software engineering problems. Section 3 describes our adaptation of NSGA-III to automate code refactoring and the results obtained from our experiment are discussed in Section 4. Related work is discussed in Section 5. Section 6 summarizes.

2. MANY-OBJECTIVE SEARCH-BASED SOFTWARE ENGINEERING

Recently many-objective optimization has attracted much attention in Evolutionary Multi-objective Optimization (EMO) which is one of the most active research areas in evolutionary computation [37]. By definition, a many-objective problem is multi-objective one but with a high number of objectives M, i.e., $M > 3$. Analytically, it could be stated as follows [8]:

$$\begin{cases} Min\ f(x) = [f_1(x), f_2(x),..., f_M(x)]^T, \quad M > 3 \\ g_j(x) \geq 0 \qquad\qquad j = 1,...,P; \\ h_k(x) = 0 \qquad\qquad k = 1,...,Q; \\ x_i^L \leq x_i \leq x_i^U \qquad i = 1,...,n. \end{cases} \tag{1}$$

where M is the number of objective functions and is *strictly greater* than 3, 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 decision variable x_i (i.e., i^{th} component of x). A solution x satisfying the (P+Q) constraints is said to be feasible and the set of all feasible solutions defines the feasible search space denoted by Ω.

In this formulation, we consider a minimization multi-objective problem (MOP) since maximization can be easily turned to minimization based on the duality principle. Over the two past decades, several Multi-Objective Evolutionary Algorithms (MOEAs) have been proposed with the hope to work with any number of objectives M. Unfortunately, It has been demonstrated that most MOEAs are ineffective in handling such type of problems. For example, NSGA-II [12], which is one of the most used MOEAs, compares solutions based on their non-domination ranks. Solutions with best ranks are emphasized in order to converge to the Pareto front. When $M > 3$, only the first rank may be assigned to every solution as almost all population individuals become non-dominated with each others. Without a variety of ranks, NSGA-II cannot keep the search pressure anymore in high dimensional objective spaces.

The difficulty faced when solving a many-objective problems could be summarized as follows. Firstly, most solutions become equivalent between each others according to the Pareto dominance relation which deteriorates *dramatically* the search process ability to converge towards the Pareto front and the MOEA behaviour becomes very similar to the random search one. Secondly, a search method requires a very high number of solutions (some thousands and even more) to cover the Pareto front when the number of objectives increases. For instance, it has been shown that that in order to find a good approximation of the Pareto front for problems involving 4, 5 and 7 objective functions, the number of required non-dominated solutions is about 62 500, 1 953 125 and 1 708 984 375 respectively [22]; which makes the decision making task very difficult. Thirdly, the objective space dimensionality increases significantly, which makes promising search directions very hard to find. Fourthly, the diversity measure estimation becomes very computationally costly since finding the neighbors of a particular solution in high dimensional spaces is very expensive. Fifthly, recombination operators becomes inefficient since population members are likely to be widely distant from each other which yields to children that are not similar to their parents; thereby making the recombination operation inefficient in producing promising offspring individuals. Finally, although it is not a matter that is directly related to optimization, the Pareto front visualization becomes more complicated, therefore making the interpretation of the MOEA's results more difficult for the user.

Recently, researchers have proposed several solution approaches to tackle many-objective optimization problems. Table 1 illustrates a summary of existing many-objective approaches. Firstly, we find the o*bjective reduction approach*, which involves finding the minimal subset of objective functions that are in conflict with each other. The main idea is to study the different conflicts between the objectives. The objective reduction approach attempts to eliminate objectives that are not essential to describe the Pareto-optimal front [37]. Even when the essential objectives are four or more, the reduced representation of the problem has a favorable impact on the search efficiency, computational cost, and decision making. However, although this approach has solved benchmark problems involving up to 20 objectives, its applicability in real world setting is not straightforward and it remains to be investigated since most objectives are usually in conflict with each other in real problems [31]. Secondly, we have the *incorporation of decision maker's preferences*: When the number of objective functions increases, the Pareto optimal approximation would be composed by a huge number of non-dominated solutions. Consequently, the selection of the final alternative would be very difficult for the human decision maker (DM). In reality, the DM is not interested with the whole Pareto front rather than the portion of the front that best matches his/her preferences, called the Region of Interest (ROI). The main idea is to exploit the DM's preferences in order to differentiate between Pareto equivalent solutions so that we can direct the search towards the ROI on problems involving more than 3 objectives [6], [5]. Preference-based MOEAs have demonstrated several promising results. Thirdly, we find *new preference ordering relations*. Since the Pareto dominance has the ability to differentiate between solutions with the increased of the number of objectives, researchers have proposed several new alternative relations. These relations try to circumvent the failure of the Pareto dominance by using additional information such as the ranks of the particular solution regarding the different objectives and the related population [14], but may not be agreeable to the decision makers. Fourthly, we have *decomposition*. This technique consists in decomposing the problem into several sub-problems and then solving these sub-problems simultaneously by exploiting the parallel search ability of evolutionary algorithms. The most reputable decomposition-based MOEA is MOEA/D [46]. Finally, we find the u*se of a predefined multiple targeted search*. Inspired by preference-based MOEAs and the decomposition approach, recently, Deb and Jain [9], [23] and Wang et al. [44] have proposed a new idea that involves guiding the population during the optimization process based on multiple predefined targets (e.g., reference points, reference direction) in the objective space. This idea

Table 1. Summary of many-objective approaches.

Approach	Basic idea	Example algorithms	No. of objectives	Real world many-objective application
Objective reduction	Find the minimal subset of conflicting objectives, then eliminate the objectives that are not essential to describe the Pareto optimal front.	1) PCA-NSGA-II [11] 2) PCSEA [41]	10 20	1) Nothing 2) Water resource problem
Incorporating decision maker's preferences	Exploit DM's preferences in order to differentiate between Pareto equivalent solutions so that we can direct the search towards the region of interest instead of the whole front.	1) r-NSGA-II [6] 2) PBEA [43] 3) R-NSGA-II [13]	10 10 10	1) Payment scheduling negotiation problem 2) Nothing 3) Welded beam design problem
New preference ordering relations	Propose alternative preference relations that are different from the Pareto dominance.	1) Preference Order Ranking–based algorithm [14] 2) Ranking dominance-based algorithm [27] 3) IBEA [47] 4) HypE [3]	8 10 5 20	1) Water distribution problem 2) Nothing 3) Software product line management 4) Nothing
Decomposition	Decompose the problem into several sub-problems and then solve these sub-problems simultaneously by exploiting the parallel search ability of EAs.	1) MOEA/D [46]	5	1) Nothing
Use of a predefined multiple targeted search	Guide the population during the optimization process based on multiple predefined targets (e.g., reference points, reference direction) in the objective space.	1) PICEA [44] 2) NSGA-III [10]	10 15	1) Nothing 2) Crash-worthiness Design of Vehicles

has demonstrated very promising results on MOPs involving up to 15 objectives.

According to a recent survey by Harman et al. [18], most software engineering problems are multi-objective by nature. However, most of existing approaches to address software engineering problems such as model transformation, design quality improvement, test suite generation, etc. are based on a mono-objective view. Multi-objective optimization techniques have been proposed in a few works [38][39][34] for such problems and they satisfy up-to 5 objectives. However, as with any other practical domain, most software engineering problems involve optimizing more than this number of objectives. Thus, more scalable search-based software engineering approaches will be beneficial to handle rich objective spaces. We investigate, in this paper, the applicability many-objective techniques for the software refactoring problem where up-to 15 objectives are considered to evaluate refactoring suggestions.

3. MANY-OBJECTIVE SOFTWARE REFACTORING USING NSGA-III

This section shows how the refactoring problem can be addressed using NSGA-III. We first present an overview of software refactoring and NSGA-III then we provide the details of our problem formulation and the solution approach.

3.1 Background

3.1.1 Software Refactoring

Refactoring is defined as the process of improving code after it has been written by changing its internal structure without changing the external behavior [17]. The idea is to reorganize variables, classes and methods to facilitate future adaptations and extensions and enhance comprehension. This reorganization is used to improve different aspects of software quality such as maintainability, extensibility, reusability, etc. Some modern integrated development environments (IDEs), such as Eclipse, NetBeans, and the Refactoring Browser [52], provide semi-automatic support for applying the most commonly used refactorings, e.g., move method, rename class, etc. However, automatically suggesting/deciding where and which refactorings to apply is still a real challenge in SE.

In order to identify which parts of the source code need to be refactored, most existing work (e.g., Fowler's textbook [17]) relies on the notion of bad smells, also called design defects, design flaws or anti-patterns. In this paper, we assume that code smells have already been detected, and need to be corrected. Typically, code smells refer to design situations that adversely affect the development of software. When applying refactorings to fix design defects, software metrics can be used as an indication of the quality of the new design. For instance, high intra-class cohesion and low inter-class coupling usually indicate a good quality system. In addition, most design defects can be detected using quality metrics (symptoms). In [25][32], the authors used quality metrics as fitness functions to evaluate suggested refactoring. However, the number of objectives was limited to one (aggregation of metrics in one objective [32]) or two (considering only two quality metrics [25]).

3.1.2 NSGA-III

NSGA-III is a very recent many-objective algorithm proposed by Deb et al. [10]. The basic framework remains similar to the original NSGA-II algorithm [12] with significant changes in its selection mechanism. Figure 1 gives the pseudo-code of the NSGA-III procedure for a particular generation *t*. First, the parent population P_t (of size *N*) is randomly initialized in the specified domain, and then the binary tournament selection, crossover and mutation operators are applied to create an offspring population Q_t. Thereafter, both populations are combined and sorted according to their domination level and the best *N* members are selected from the combined population to form the parent population for the next generation. The fundamental difference between NSGA-II and NSGA-III lies in the way the niche preservation operation is performed. Unlike NSGA-II, NSGA-III starts with a set of reference points Z^r. After non-dominated sorting, all acceptable front members and the last front F_l that could not be completely accepted are saved in a set S_t. Members in S_t/F_l are selected right away for the next generation. However, the remaining members are selected from F_l such that a desired diversity is maintained in the population. Original NSGA-II uses the crowding distance measure for selecting well-distributed set of points, however, in NSGA-III the supplied reference points (Z^r) are used to select these remaining members (cf. Figure 2). To accomplish this, objective values and reference points are first normalized so that they have an identical range. Thereafter, orthogonal distance between a member in S_t and each of the reference lines (joining the ideal point and a reference point) is computed. The member is then associated with the reference point having the smallest orthogonal distance. Next, the niche count ρ for each reference point, defined as the number of members in S_t/F_l that are associated with the reference point, is computed for further processing. The reference point having the minimum niche count is identified and the member from the last front F_l that is associated with it is included in the final population. The niche count of the identified reference point is increased by one and the procedure is repeated to fill up population P_{t+1}.

It is worth noting that a reference point may have one or more population members associated with it or need not have any population member associated with it. Let us denote this niche count as ρ_j for the *j*-th reference point. We now devise a new niche-preserving operation as follows. First, we identify the reference point set $J_{\min} = \{j\text{: argmin}_j\ (\rho_j)\}$ having minimum ρ_j. In case of multiple such reference points, one ($j^* \in J_{\min}$) is chosen at random. If $\rho_{j^*} = 0$ (meaning that there is no associated P_{t+1} member to the reference point j^*), two scenarios can occur. First, there exists one or more members in front F_l that are already associated with the reference point j^*. In this case, the one having the shortest perpendicular distance from the reference line is added to P_{t+1}. The count ρ_{j^*} is then incremented by one. Second, the front F_l does not have any member associated with the reference point j^*. In this case, the reference point is excluded from further consideration for the current generation. In the event of $\rho_{j^*} \geq 1$ (meaning that already one member associated with the reference point exists), a randomly chosen member, if exists, from front F_l that is associated with the reference point F_l is added to P_{t+1}. If such a member exists, the count ρ_{j^*} is incremented by one. After ρ_j counts are updated, the procedure is repeated for a total of *K* times to increase the population size of P_{t+1} to *N*.

NSGA-III procedure at generation *t*

Input: *H* structured reference points Z^s, parent population P_t
Output: P_{t+1}

Begin
$S_t \leftarrow \emptyset$, $i \leftarrow 1$;
$Q_t \leftarrow$ Variation (P_t);
$R_t \leftarrow P_t \cup Q_t$;
$(F_1, F_2, \ldots) \leftarrow$ Non-dominationed_Sort (R_t);
Repeat
  $S_t \leftarrow S_t \cup F_i$; $i \leftarrow i+1$;
Until $|S_t| \geq N$;
$F_l \leftarrow F_i$; /*Last front to be included*/
If $|S_t| = N$ **then**
  $P_{t+1} \leftarrow S_t$;
Else
  $P_{t+1} \leftarrow \bigcup_{j=1}^{l-1} F_j$;
  /*Number of points to be chosen from F_l*/
  $K \leftarrow N - |P_{t+1}|$;
  /*Normalize objectives and create reference set Z^r*/
  Normalize (F^M; S_t; Z^r; Z^s);
  /*Associate each member *s* of S_t with a reference point*/
  /*$\pi(s)$: closest reference point*/
  /*$d(s)$: distance between *s* and $\pi(s)$*/
  $[\pi(s), d(s)] \leftarrow$ Associate (S_t, Z^r);
  /*Compute niche count of reference point $j \in Z^r$ */
  $\rho_j \leftarrow \sum_{s \in S_t / F_l} ((\pi(s) = j)\ ?\ 1 : 0)$;
  /*Choose *K* members one at a time from F_l to construct P_{t+1}*/
  Niching (K, ρ_j, $\pi(s)$, $d(s)$, Z^r, F_l, P_{t+1});
End If
End

Figure 1. Pseudocode of NSGA-III main procedure.

3.2 Adapting NSGA-III for the Software Refactoring Problem

3.2.1 *Problem formulation*

The refactoring problem involves searching for the best refactoring solution among the set of candidate ones, which constitutes a huge search space. A refactoring solution is a sequence of refactoring operations where the goal of applying the sequence to a software system *S* is typically to minimize the number of code smells in *S*. Usually in SBSE approaches, we use two or three metrics as objective functions for a particular multi-objective heuristic algorithm to find smells and correct them. However, in reality, there are many types of code smell and detecting the symptoms of each smell requires a particular set of metrics. Motivated by this observation, we propose in this research work to use a high number of metrics (15 metrics) where each represents a separate objective function. In this way, we obtain a many-objective (15-objective) formulation of the refactoring problem that could not be solved using standard multi-objective approaches. This formulation is given as follows:

$$\begin{aligned} &\textit{Maximize} \quad F(x,S) = [f_1(x,S), f_2(x,S), \ldots, f_{15}(x,S)] \\ &\textit{subject to} \quad x = (x_1, \ldots, x_n) \in X \end{aligned}$$

where *X* is the set of all legal refactoring sequences starting from *S*, x_i is the *i*-th refactoring operation, and $f_k(x,S)$ is the *k*-th metric. The 15 metrics under consideration will be detailed in the experimental study since our formulation is generic and applies to any software metrics.

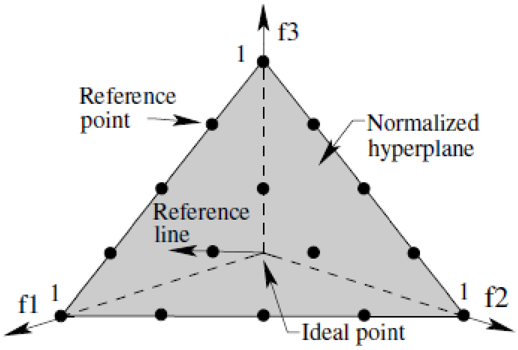

Figure 1. Normalized reference plane for a three-objective case [19].

3.2.2 Solution approach

In this section, we describe our adaptation of NSGA-III to the problem of software refactoring. As noted by Harman et al. [20], the use of search algorithms to solve SE problems requires some adaptation steps. For NSGA-III, these steps are: (1) solution representation, (2) solution variation, (3) solution evaluation, and (4) normalization of refactoring sequences.

Solution representation. As defined in the previous section, a solution consists of a sequence of *n* refactoring operations applied to different code elements in the source code to fix. In order to represent a candidate solution (individual/chromosome), we use a vector-based representation. Each vector's dimension represents a refactoring operation where the order of applying these refactoring operations corresponds to their positions in the vector. For each of these refactoring operations, we specify pre- and post-conditions in the style of Opdyke [33] to ensure the feasibility of their application. The initial population is generated by assigning randomly a sequence of refactorings to some code fragments. To apply a refactoring operation we need to specify which actors, i.e., code fragments, are involved/impacted by this refactoring and which roles they play in performing the refactoring operation. An actor can be a package, class, field, method, parameter, statement, or variable.

Solution variation. In each search algorithm, the variation operators play the key role of moving within the search space with the aim of driving the search towards optimal solutions. For crossover, we use the one-point crossover operator. It starts by selecting and splitting at random two parent solutions. Then, this operator creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and vice versa for the second child. This operator must ensure the respect of the length limits by eliminating randomly some refactoring operations. It is important to note that in many-objective optimization, it is better to create children that are close to their parents in order have a more efficient search process [10][36]. For this reason, we control the cutting point of the one-point crossover operator by restricting its position to be either belonging to the first tier of the refactoring sequence or belonging to the last tier. For mutation, we use the bit-string mutation operator that picks probabilistically one or more refactoring operations from its or their associated sequence and replace them by other ones from the initial list possible refactorings.

Solution evaluation. Each generated refactoring solution is executed on the system *S*. Once all required data is computed, the solution is evaluated based on the 15 metrics used as objective functions. Based on these values, the refactoring solution is assigned a non-domination rank (as in NSGA-II) and a position in the objective space allowing it to be assigned to a particular reference point based on distance calculation as previously described.

Normalization of population members. Usually objective functions (metrics) are incommensurable (i.e., they have different scales). For this reason, we used the normalization procedure proposed by Deb et al. [10] to circumvent this problem. At each generation, the minimal and maximal values for each metric are recorded and then used by the normalization procedure. Normalization allows the population members and with the reference points to have the same range, which is a pre-requisite for diversity preservation.

4. DESIGN OF THE EMPIRICAL STUDY

In order to evaluate our approach for fixing code smells using NSGA-III, we conducted a set of experiments based on different versions of large open source systems [48][54][49][50][51]. Each experiment is repeated 31 times, and the obtained results are subsequently statistically analyzed with the aim to compare our NSGA-III proposal with a variety of existing approaches [12][46][25][32][47]. In this section, we first present our research questions and then describe and discuss the obtained results. Finally, we discuss the various threats to the validity of our experiments.

4.1 Research Questions

The study was conducted to assess the applicability of our many-objective refactoring approach when applied in real-world settings and to compare its performance with state of the art approaches. More specifically, we aimed to answer the following research questions:

RQ1: How does NSGA-III perform compared to other many-objective (MOAE/D [46], IBEA [47]) and multi-objective (NSGA-II [12]) techniques? It is important to evaluate the performance of NSGA-III in terms of scalability when the number of considered objectives increases. In addition, it is interesting to determine if considering more metrics (objectives) improves the quality of the suggested refactoring solutions (the number of fixed code smells).

RQ2: How does NSGA-III perform compared to mono-objective refactoring approaches [25][32]? It is important to determine if considering each conflicting metric as a separate objective to optimize performs better than a mono-objective approach that aggregates all metrics in one objective. The comparison between a many-objective EA with a mono-objective one is not straightforward. The first one returns a set of non-dominated solutions while the second one returns a single optimal solution. In order to resolve this problem, for each many-objective algorithm we choose the nearest solution to the ideal point [4] (i.e., the vector composed of the best objective values among the population members) as a candidate solution to be compared with the single solution return by the mono-objective algorithm.

RQ3: How does our many-objective formulation scale? There is a cost in allowing the developer to specify a large number of objectives. Can it be demonstrated that as the number of objectives increases, we can achieve a commensurate increase in the quality of the solutions generated? If not, then our approach is not justified.

4.2 Experimental Setup

4.2.1 Systems Studied

Our study considers the extensive evolution of different open source Java systems analyzed in the literature [25][34][30][29][35]. The corpus used includes releases of Apache Ant [48], ArgoUML [49], Gantt [51], Azureus [50] and Xerces-J [54]. 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 700 in total. Indeed, in several works [25][34][30][29][35], 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 purposes, they provided a corpus describing instances of different code smells including blob, spaghetti code, and functional decomposition [17]. In our study, we verified the capacity of our approach to fix classes that correspond to instances of these code smells.

We choose the above-mentioned open source systems because they are medium/large-sized open-source projects and were analyzed in related work. The initial versions of Apache Ant were known to be of poor quality, which has led to major revised versions. Xerces-J, ArgoUML, Azureus and Gantt have all been actively developed over the past 10 years, and their design has not been responsible for a slowdown in their developments. We used the detection rules of code smells proposed in Kessentini et al. [25] to identify the number of fixed code smells after applying the best refactoring solutions.

Table 2. Features of software systems analyzed.

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	41
Azureus v2.3.0.6	1449	108

4.2.2 Performance Indicators

We used mainly three performance indicators to compare the different algorithms used in our experiments. These indicators are defined as follows:

- *Inverted Generational Distance (IGD):* A number of performance metrics for multi-objective optimization have been proposed and discussed in the literature, which aim to evaluate the closeness to the Pareto optimal front and the diversity of the obtained solution set, or both criterion. Most of the existing metrics require the obtained set to be compared against a specified set of Pareto optimal reference solutions. In this study, the inverted generational distance (IGD) is used as the performance metric since it has been shown to reflect both the diversity and convergence of the obtained non-dominated solutions [10]. The IGD corresponds to the average Euclidean distance separating each reference solution from its closest non-dominated one. Note that for each system we use the set of Pareto optimal solutions generated by all algorithms over all runs as reference solutions.

- *Percentage of fixed code smells (NF)* is the percentage of code smells fixed by the application of the best refactoring solution (i.e., number of fixed smells divided by the total number of code smells). The detection of code smells after applying a refactoring solution is performed using the detection rules of Kessentini et al. [25].

- *Computational time (CT)* is used mainly to compare the efficiency of NSGA-III with other algorithms using the same number of objectives.

4.2.3 Statistical Tests

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is performed based on 31 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test [2] with a 95% confidence level (α = 5%). The latter verifies the null hypothesis H_0 that the obtained results of two algorithms are samples from continuous distributions with equal medians, against the alternative that they are not H_1. The *p*-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H_0 while it is true (type I error). A *p*-value that is less than or equal to α ($\leq$ 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. In fact, for each problem instance, we compute the *p*-value obtained by comparing NSGA-II, IBEA, MOEA/D and mono-objective search results with NSGA-III ones. In this way, we determine whether the performance difference between NSGA-III and one of the other approaches is statistically significant or just a random result.

To answer RQ1, We compared the performance of NSGA-III with two many-objective techniques, MOEA/D [46] and IBEA [47], and also with a multi-objective algorithm that uses NSGA-II [12]. The obtained results were compared using *IGD* and *NF*.

To answer RQ2, we compared NSGA-III with two mono-objective refactoring approaches where an aggregation of quality metrics is used to form one objective. Kessentini et al. [25] use genetic algorithms to find the best sequence of refactorings that minimizes the number of code smells while O'Keeffe et al. [32] use different mono-objective algorithms to find the best sequence of refactorings that optimize a fitness function composed of a set of quality metrics.

To answer RQ3, we compared different executions of NSGA-III and other many-objective algorithms using varying numbers of objectives: 3, 5, 8, 10 and 15 to demonstrate that considering higher number of objectives (metrics) can improve the quality of refactoring results.

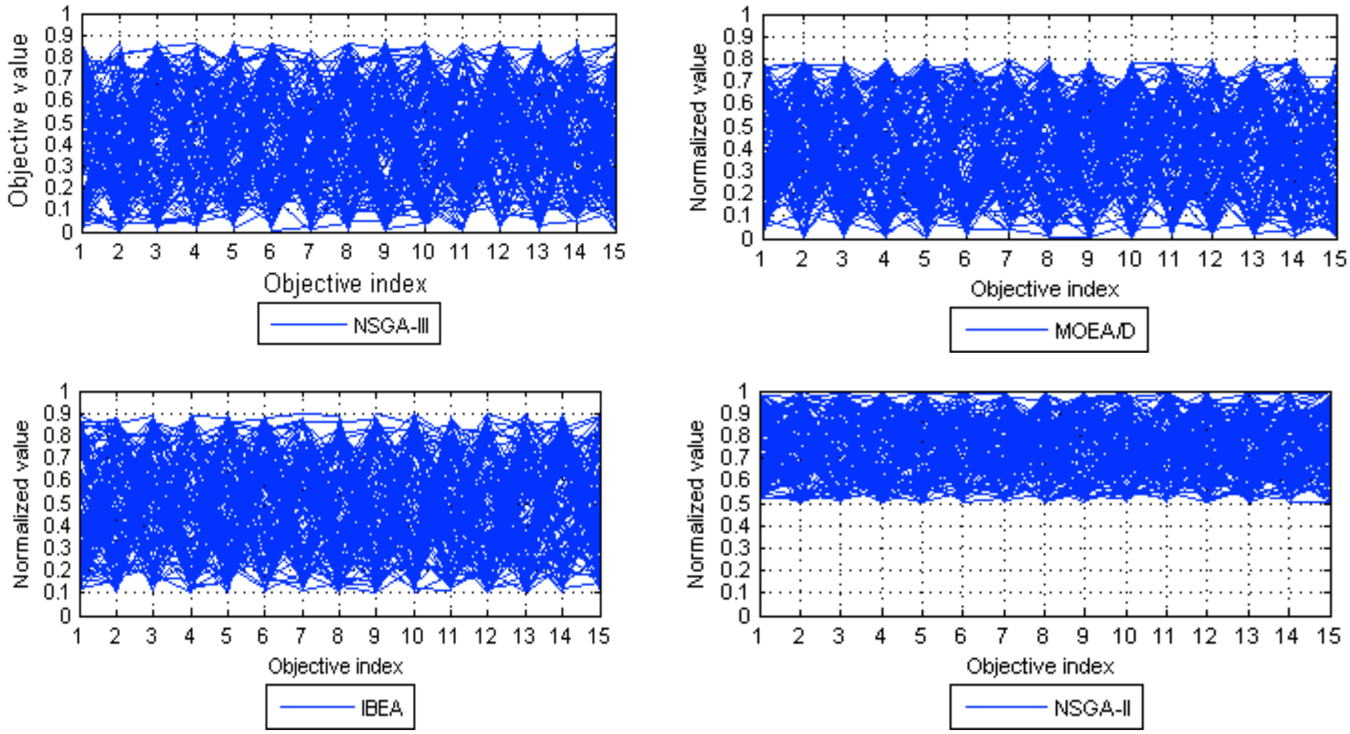

Figure 2. Value path plots of non-dominated solutions obtained by NSGA-III, MOEA/D, IBEA and NSGA-II during the median run of the 15-objective refactoring problem on ArgoUML v0.26.

4.2.4 Parameter Settings

Parameter setting influences significantly the performance of a search algorithm on a particular problem [1]. For this reason, for each many-objective algorithm and for each system (cf. Table 3), we perform a set of experiments using several population sizes: 91, 210, 156, 275 and 135 for respectively 3, 5, 8, 10 and 15 objectives. The maximum number of generations used is 400, 600, 750, 1000 and 1500 respectively for 3, 5, 8, 10 and 15 objectives. Each algorithm is executed 31 times with each configuration and then comparison between the configurations is done based on *IGD* using the Wilcoxon test. In order to have significant results, for each couple (algorithm, system), we use the trial and error method [28] in order to obtain a good parameter configuration. Since we are comparing different search algorithms, we classify parameters into common parameters and specific parameters. Table 3 depicts the important common parameters. We used a set of 15 quality metrics, namely Weighted Methods per Class (WMC), Response for a Class (RFC), Lack of Cohesion of Methods (LCOM), Cyclomatic Complexity (CC), Number of Attributes (NA), Attribute Hiding Factor (AH), Method Hiding Factor (MH), Number of Lines of Code (NLC), Coupling Between Object Classes (CBO), Number of Association (NAS), Number of Classes (NC), Depth of Inheritance Tree (DIT), Polymorphism Factor (PF), Attribute Inheritance Factor (AIF) and Number of Children (NOC) [15]. We selected randomly at each run some metrics from this list when the number of objectives is lower than 15. We used 23 refactoring types in our experiments, namely Add Parameter, Rename Method Encapsulate Collection/Downcast/Field, Collapse Hierarchy, Hide Method, Extract Class/Interface/Method/Subclass/Superclass, Inline Class/Method, Move Field/Method, Pull Up Field/Method, Push Down Field/Method and Remove Parameter/Setting Method [53].

Table 3. The setting of common parameters.

Number of objectives	Number of reference points (for NSGA-III and MOEA/D)	Population size
3	91	91
5	210	210
8	156	156
10	275	275
15	135	135

4.3 Results

Table 4 shows the median IGD and NF values over 31 independent runs for all algorithms under comparison. All the results were statistically significant on the 31 independent simulations using the Wilcoxon rank sum test [2] with a 99% confidence level ($\alpha < 1\%$). For the 3-objective case, we see that NSGA-III and NSGA-II present similar results, and that NSGA-III provides slightly better results than IBEA and MOEA/D. For the 5-objective case, NSGA-III strictly outperforms NSGA-II and gives similar results to those of the two other multi-objective algorithms. For the 8-objective case, NSGA-III is strictly better than NSGA-II and significantly better than IBEA and MOEA/D. Additionally, IBEA seems to be slightly better than MOEA/D. It is worth noting that for problems instances with more 8 objectives or more, NSGA-II performance is dramatically degraded, which is simply denoted by the ~ symbol. For the 10- and 15-objective case, NSGA-III is strictly better than all other algorithms. Moreover, MOEA/D seems to significantly outperform IBEA. The performance of NSGA-III could be explained by the interaction between: (1) Pareto dominance-based selection and (2) reference point-based selection, which is the distinguishing feature of NSGA-III compared to other existing many-objective algorithms.

Table 4. Median IGD and NF values on 31 runs (best values are in bold). ~ means a large value that is not interesting to show. The results were statistically significant on 31 independent runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).

Problem	*M*	MaxGen	NSGA-III		IBEA		MOEA/D		NSGA-II	
			NF	IGD	NF	IGD	NF	IGD	NF	IGD
ArgoUML v0.26	3	400	**69%**	**1.356×10^{-3}**	67%	1.358×10^{-3}	**69%**	1.359×10^{-3}	67%	**1.356×10^{-3}**
	5	600	**83%**	**3.918×10^{-3}**	79%	4.001×10^{-3}	81%	4.007×10^{-3}	54%	4.423×10^{-3}
	8	750	**86%**	**4.115×10^{-3}**	82%	4.327×10^{-3}	84%	4.331×10^{-3}	42%	~
	10	1000	**89%**	**1.727×10^{-2}**	84%	2.124×10^{-2}	87%	2.001×10^{-2}	37%	~
	15	1500	**96%**	**3.302×10^{-2}**	81%	3.826×10^{-2}	84%	3.733×10^{-2}	34%	~
Xerces v2.7	3	400	**64%**	**9.751×10^{-4}**	**64%**	9.754×10^{-4}	**64%**	9.753×10^{-4}	62%	9.752×10^{-4}
	5	600	**78%**	**7.876×10^{-3}**	76%	7.910×10^{-3}	74%	7.912×10^{-3}	56%	8.006×10^{-3}
	8	750	**84%**	**8.001×10^{-3}**	81%	8.422×10^{-3}	81%	8.428×10^{-3}	48%	~
	10	1000	**92%**	**2.115×10^{-2}**	86%	2.410×10^{-2}	86%	2.299×10^{-2}	44%	~
	15	1500	**94%**	**4.666×10^{-2}**	86%	5.198×10^{-2}	84%	4.935×10^{-2}	39%	~
ArgoUML v0.3	3	400	63%	**2.667×10^{-3}**	63%	2.668×10^{-3}	**66%**	2.668×10^{-3}	63%	**2.667×10^{-3}**
	5	600	**79%**	**4.283×10^{-3}**	76%	4.291×10^{-3}	78%	4.294×10^{-3}	51%	4.524×10^{-3}
	8	750	**91%**	**5.545×10^{-3}**	84%	5.701×10^{-3}	86%	5.716×10^{-3}	47%	~
	10	1000	**94%**	**3.339×10^{-2}**	87%	3.601×10^{-2}	89%	3.477×10^{-2}	39%	~
	15	1500	**98%**	**6.001×10^{-2}**	89%	6.554×10^{-2}	87%	6.399×10^{-2}	34%	~
Ant-Apache v1.5	3	400	**68%**	**3.854×10^{-4}**	66%	3.857×10^{-4}	66%	3.856×10^{-4}	**68%**	3.856×10^{-4}
	5	600	76%	**4.678×10^{-4}**	76%	4.702×10^{-4}	**78%**	4.709×10^{-4}	52%	5.035×10^{-4}
	8	750	**88%**	**6.111×10^{-4}**	81%	6.308×10^{-4}	83%	6.313×10^{-4}	46%	~
	10	1000	**93%**	**2.861×10^{-3}**	84%	3.008×10^{-3}	86%	2.999×10^{-3}	34%	~
	15	1500	**97%**	**5.928×10^{-3}**	82%	6.478×10^{-3}	81%	6.399×10^{-3}	31%	~
Ant-Apache v1.7.0	3	400	**61%**	**4.326×10^{-3}**	61%	4.328×10^{-3}	63%	4.329×10^{-3}	**62%**	4.327×10^{-3}
	5	600	**73%**	**5.612×10^{-3}**	69%	5.692×10^{-3}	71%	5.694×10^{-3}	53%	6.001×10^{-3}
	8	750	**86%**	**6.269×10^{-3}**	84%	6.516×10^{-3}	82%	6.521×10^{-3}	48%	~
	10	1000	**89%**	**4.007×10^{-2}**	87%	4.301×10^{-2}	84%	4.147×10^{-2}	37%	~
	15	1500	**92%**	**7.006×10^{-2}**	84%	7.840×10^{-2}	81%	7.688×10^{-2}	31%	~
Gantt v1.10.2	3	400	63%	**5.111×10^{-3}**	64%	5.113×10^{-3}	63%	5.114×10^{-3}	**66%**	**5.111×10^{-3}**
	5	600	**69%**	**6.601×10^{-3}**	67%	6.702×10^{-3}	**69%**	6.701×10^{-3}	54%	6.956×10^{-3}
	8	750	**88%**	**7.899×10^{-3}**	83%	8.101×10^{-3}	81%	8.116×10^{-3}	44%	~
	10	1000	**93%**	**3.333×10^{-2}**	84%	3.637×10^{-2}	86%	3.536×10^{-2}	37%	~
	15	1500	**97%**	**5.807×10^{-2}**	82%	6.202×10^{-2}	83%	6.125×10^{-2}	28%	~
Azureus v2.3.0.6	3	400	61%	**6.429×10^{-4}**	64%	6.432×10^{-4}	61%	6.431×10^{-4}	**64%**	6.431×10^{-4}
	5	600	**78%**	**6.708×10^{-4}**	72%	6.782×10^{-4}	75%	6.788×10^{-4}	47%	6.991×10^{-4}
	8	750	**84%**	**6.976×10^{-4}**	80%	7.205×10^{-4}	82%	7.212×10^{-4}	34%	~
	10	1000	**91%**	**2.745×10^{-3}**	84%	2.976×10^{-3}	87%	2.877×10^{-3}	26%	~
	15	1500	**94%**	**4.981×10^{-3}**	81%	5.508×10^{-3}	78%	5.394×10^{-3}	26%	~

The percentage of fixed code smells using NSGA-III is better than all other algorithms in all systems in 100% of cases when more than eight objectives are considered. It is clear from Table 4 that the percentage of fixed code smells increases as we use more quality metrics to evaluate refactoring solutions. On average, all the four algorithms NSGA-III, IBEA, MOEA/D and NSGA-II perform similarly with 3 objectives, however the percentage of code smells fixed is low in all systems. This is due to the fact that the use of only three quality metrics is not enough to evaluate the quality of the design after applying the best refactoring solution. The average percentage of fixed code smells in all systems using NSGA-III with 15 objectives on all systems is higher than 92%, which outperforms all the remaining algorithms. Thus, we can conclude that NSGA-III represents a scalable solution to find trade-offs between 15 objectives and that the use of additional objectives (metrics) improves the quality of refactoring solutions.

Figure 2 illustrates the value path plots of all algorithms the 15-objective refactoring problem on ArgoUMLv0.26, the largest system used in our experiments. Similar observations were made in the remaining systems but are omitted due to space considerations. All quality metrics were normalized between 0 and 1 and all are to be minimized. We observe that NSGA-III presents the best convergence since its non-dominated solutions are the closest to the ideal point, i.e., the vector composed of 15 zeros. Also, MOEA/D seems to have better convergence than IBEA. However, NSGA-II is unable to progress in terms of convergence as its non-dominated solutions are so far from the ideal vector. We conclude that although NSGA-II is the most famous multi-objective algorithm in SBSE, it is not adequate for problems involving more than 3 objectives. Based on the results we obtained for the refactoring problem, it appears that NSGA-III is a very good candidate solution for tackling many-objective SBSE problems.

We compared also the results of NSGA-III using 15 objectives and two mono-objective refactoring approaches on all seven open source systems as described in Figure 3. From the set of non-dominated solutions generated by NSGA-III, we selected the solution closest to the ideal point. NSGA-III performed better than both mono-objective algorithms in 100% of cases. In fact, since mono-objective algorithms aggregate all metrics in one objective there is a loss of information due to the conflicting nature of the used quality metrics. Especially in the case of large systems such as Ant Apache and Argo UML, it is clear that mono-objective algorithms did not perform well in terms of fixing code smells whereas NSGA-III fixed on average more than 92% of them. In general, large systems contain many different types of code smells and so a large number of metrics is required to evaluate the quality of a system after applying a refactoring solution.

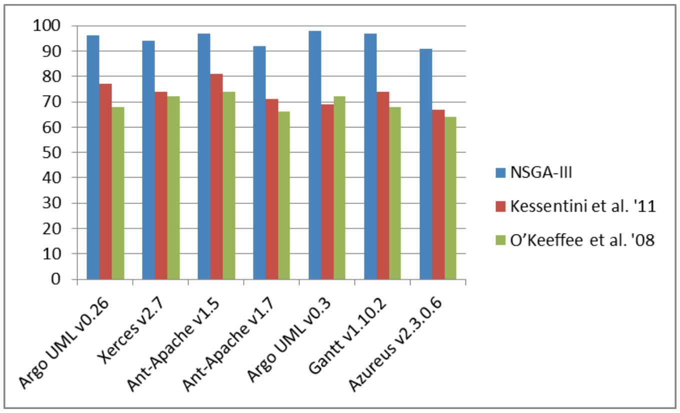

Figure 3. Median NF values on 31 independent runs using 7 systems comparing NSGA-III and two mono-objective approaches [25][32]

4.4 Discussion

Computational time (CT):

When using optimization techniques, the most time consuming operation is the evaluation step [42]. Thus, we studied the execution time of all many/multi-objective algorithms used in our experiments. Figure 4 shows the evolution of the running times of the different algorithms on the ArgoUMLv0.26 system, the largest system in our experiments. It is clear from this figure, that the multi-objective algorithm (NSGA-II) has similar running times for the 3- and 5-objective cases. However, for higher number of objectives NSGA-III is faster than IBEA. This observation could be explained by the computational effort required to compute the contribution of each solution in terms of hypervolume. In comparison to MOEA/D, MOEA/D is slightly faster than NSGA-III since it does not make use of non-dominated sorting.

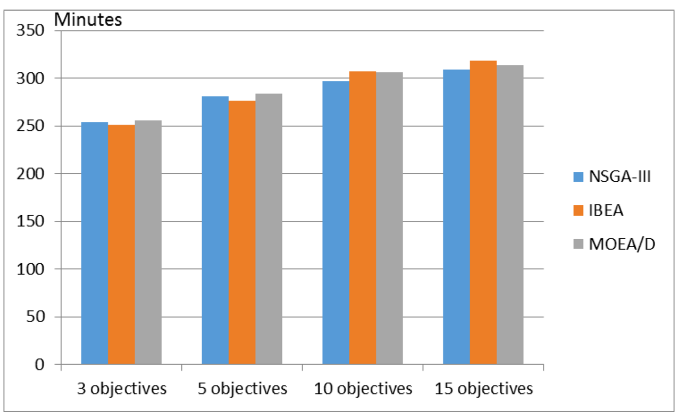

Figure 4. Median CT values on 31 independent runs using 7 systems comparing NSGA-III and two mono-objective approaches.

Quality improvements vs. number of objectives:

One of the main motivations of our work is to propose a scalable search-based software engineering approach that can address software engineering problems with a large number of objectives to be optimized. Thus, we evaluated the impact of taking into consideration a higher number of objectives (metrics) on the quality of the refactoring solutions. In fact, the symptoms of code smells can be formalized in terms of quality metrics, thus if we consider more metrics in evaluating a refactoring solution there is better chance that more code smells are fixed. Figure 5 confirms this. The percentage of fixed code smells increases from 63% to 98% as the number of

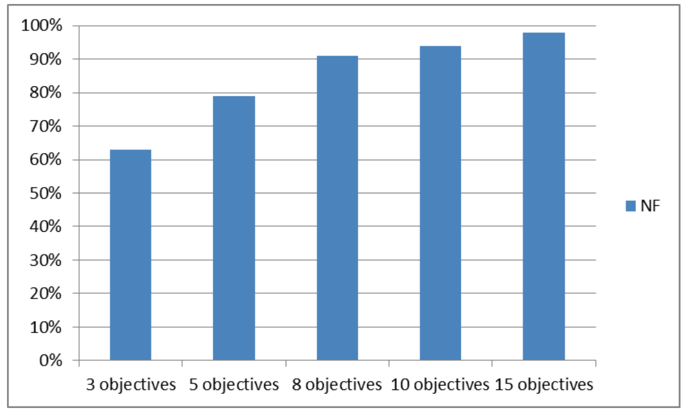

Figure 5. Median NF values on 31 independent runs obtained by NSGA-III using ArgoUML v 0.3 with 3, 5, 8, 10 and 15 objectives

objectives/metrics increases from 3 to 15 objectives. This result allows us to affirm RQ3.

Size of the solution vs. number of objectives:

The size of a refactoring solution is also important. It is usual that developers prefer the solution that fixes the maximum number of code smells with a minimum number of refactoring. A smaller number of refactoring will reduce the number of code changes required, thus it is interesting to study the size of the best refactoring solutions proposed by the different algorithms. Figure 6 describes the median number of refactorings on 31 independent runs using ArgoUML v 0.3 produced by NSGA-III, IBEA, MOEA/D and NSGA-II with 3, 5, 10 and 15 objectives. The figure shows that the number of refactorings increases when we increase the number of objectives/metrics. For example, using NSGA-III the number of refactorings increases from 178 to 234 as the number of metrics increases from 3 to 15. The simplest explanation is that as the number of objectives increases more code smells are fixed, and this requires additional refactorings. Another factor is that finding trade-offs between a higher number of metrics requires a higher number of refactoring to be applied. Another interesting observation is that NSGA-III provides smaller refactoring solutions than IBEA and MOEA/D with 3, 5, 10 and 15 objectives. Thus, our NSGA-III formulation finds also a good trade-off between maximizing the number of fixed code smells and minimizing the number of used refactorings.

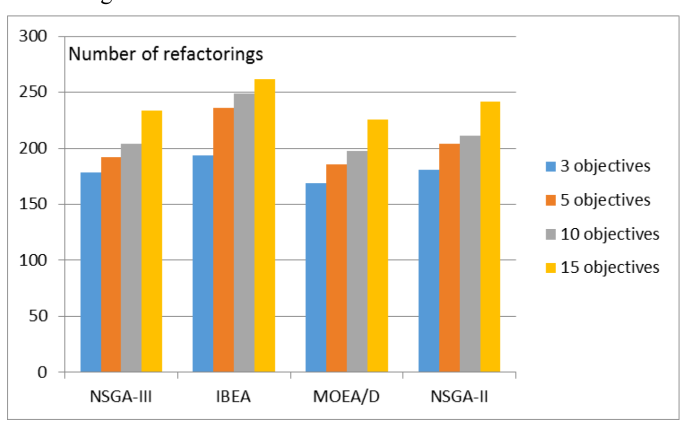

Figure 6. Median number of refactorings on 31 independent runs using ArgoUML v 0.3 produced by NSGA-III, IBEA, MOEA/D and NSGA-II with 3, 5, 10 and 15 objectives.

4.5 Threats to validity

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

In our experiments, construct validity threats are related to the absence of similar work that uses many-objective techniques for software refactoring. For that reason we compare our proposal with two other many-objective techniques [46][47] using the same objectives. Another threat to construct validity arises because, although we considered 15 widely-used quality metrics, we did not evaluate the use of other quality metrics. In future work, we plan to use additional quality metrics and compare the results with our current proposal. Another construct threat can be related to the corpus of manually detected code smells since developers do not all agree if a candidate is a code smell or not. We will ask some new experts to extend the existing corpus and provide additional feedback regarding the detected code smells.

We take into consideration the internal threats to validity in the use of stochastic algorithms since our experimental study is performed based on 31 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test [2] 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.

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

Several studies have recently focused on software refactoring using a variety of techniques. These techniques range from fully automatic to manual refactoring approaches. In this section, we focus mainly on search-based refactoring approaches since other refactoring work has already been discussed in Section 3.1. We discuss also briefly the one existing work related to the use of many-objective techniques to address a software engineering problem.

Search-based refactoring, i.e., fully automated refactoring driven by metaheuristic search and guided by software quality metrics and used subsequently to address the problem of automating design improvement [32]. Seng et al. [40] propose a search-based technique that uses a genetic algorithm over refactoring sequences. The employed metrics are mainly related to various class level properties such as coupling, cohesion, complexity and stability. The approach was limited only to the use of one refactoring operation type, namely 'move method'. In contrast to O'Keeffe et al. [32], their fitness function is based on well-known measures of coupling between program components. Both these approaches use weighted-sum to combine metrics into a fitness function, which is of practical value but is a questionable operation on ordinal metric values. Kessentini et al. [25] also propose a single-objective combinatorial optimization using a genetic algorithm to find the best sequence of refactoring operations that improve the quality of the code by minimizing as much as possible the number of code smells detected using a set of quality metrics. Kilic et al. explore the use of a variety of population-based approaches to search-based parallel refactoring, finding that local beam search could find the best solutions [26].

Harman and Tratt were the first to introduce the concept of Pareto optimality to search-based refactoring [19]. They use it to combine two metrics into a fitness function, namely CBO (coupling between objects) and SDMPC (standard deviation of methods per class), and demonstrate that it has several advantages over the weighted-sum approach. More recent work on multi-objective search-based refactoring is the work by Ouni et al. [34] who propose a multi-objective optimization approach to find the best sequence of refactorings using NSGA-II. The proposed approach is based on two objective functions, quality (proportion of corrected code smells) and code modification effort, to recommend a sequence of refactorings that provide the best trade-off between quality and effort. It is worth noting that search-based refactoring has applications other than improving design quality and correcting code smells. However, most existing search-based refactoring work is based on the use of quality metrics to evaluate suggested refactoring solutions. Our work offers a scalable approach that considers a large number of quality metrics to improve refactoring results. To the best of our knowledge, only one other work uses a many-objective technique to address a software engineering problem, namely the work of Abdel Salam et al. [38][39], who propose the use of an Indicator-Based Evolutionary Algorithm [47] for configuring software product lines. Five objectives are optimized in order to find the best configuration.

In this paper, we propose the first scalable, many-objective search-based software engineering approach that supports the optimization of 15 objectives. In addition, our work represents the first real-world application of NSGA-III proposed recently by K.Deb et al. [10].

6. CONCLUSIONS AND FUTURE WORK

In this paper we introduced a novel, scalable, search-based software engineering approach based on NSGA-III that is capable of supporting the use of 15 objectives in exploring the search space. In the first real-world application of NSGA-III, we applied our approach to the software refactoring problem, using a set of 15 software quality metrics to evaluate the proposed refactoring solutions. We evaluated our approach on seven large open source systems. The experimental results indicate that NSGA-III outperforms other many-objective algorithms (namely IBEA [47] and MOEA/D [46]), NSGA-II and mono-objective evolutionary algorithms [25][32]. All results were found to be statistically significant over 31 independent runs using the Wilcoxon rank sum test [41] with a 99% confidence level ($\alpha < 1\%$). More than 92% of the code smells were fixed on the various open source systems that were evaluated, thus demonstrating that our NSGA-III approach represents the new state of the art in fully-automated software refactoring.

As part of future work, we plan to adapt NSGA-III to apply it to other complex software engineering problems that hitherto could not be easily addressed using a many-objective approach. In the context of the software refactoring problem, we envisage using non-structural metrics (semantic-based metrics, information from software repositories, etc.) to enhance the refactoring process. We will also perform further comparative studies on large open source systems, and will investigate the impact of different parameter settings on the quality of our results. In any case, the extensive study presented in this paper has shown that NSGA-III can handle as many as 15 objectives in the context of solving software engineering problems.

7. REFERENCES

[1] Arcuri A. and Fraser G., 2013. *Parameter tuning or default values? An empirical investigation in search-based software engineering*, Empirical Software Engineering, 18(3).

[2] Arcuri, A. and 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 '11). ACM, New York, NY, USA, 1-10. DOI=10.1145/1985793.1985795.

[3] Bader, J. and Zitzler, E. 2011. Hype: An algorithm for fast hypervolumebased many-objective optimization. In *Evolutionary Computation*. vol. 19, no. 1. 45–76.

[4] Bechikh, S., Ben Said, L. and Ghédira, K. 2010. Estimating Nadir Point in Multi-objective Optimization using Mobile Reference Points. In *Proceedings of the IEEE Congress on Evolutionary Computation* (CEC'10). 2129–2137.

[5] Bechikh, S., Ben Said, L. and Ghédira, K. 2011. Searching for Knee Regions of the Pareto Front using Mobile Reference Points. Soft Computing. *A Fusion of Foundations, Methodologies and Applications*. Vol. 15, no. 9. 1807–1823.

[6] Ben Said, L., Bechikh, S., and Ghédira, K. 2010. The r-Dominance: A New Dominance Relation for Interactive Evolutionary Multicriteria Decision Making. In *Proceedings of IEEE Transactions on Evolutionary Computation*. vol. 14, no. 5. 801–818.

[7] Brown, W. J., Malveau, R. C., Brown, W. H., and Mowbray, T. J. 1998. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1st ed.

[8] Deb, K. 2001. *Multiobjective Otpimization using Evolutionary Algorithms*. John Wiley and Sons, Ltd, New York, USA.

[9] Deb, K. and Jain H. 2012. Handling many-objective problems using an improved NSGA-II procedure. In *Proceedings of IEEE Congress on Evolutionary Computation*. 1–8.

[10] Deb, K. and Jain, H. An Evolutionary Many-Objective Optimization Algorithm Using Reference-point Based Non-dominated Sorting Approach, Part I: Solving Problems with Box Constraints. In *Proceedings of IEEE Transactions on Evolutionary Computation*. accepted.

[11] Deb, K. and Saxena, D. K. 2006. Searching for pareto-optimal solutions through dimensionality reduction for certain large-dimensional multiobjective optimization problems. In *Proceedings of IEEE Congress on Evolutionary Computation*. 3353–3360.

[12] Deb, K., Pratap, A., Agarwal, S. and Meyarivan, T. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *In Proceedings of. IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2. 182–197.

[13] Deb, K., Sundar, J., Uday, N. and Chaudhuri, S. 2006. Reference point based multi-objective optimization using evolutionary algorithms. In *International Journal of Computational Intelligence Research* (IJCIR'06). vol. 2, no. 6, 273–286.

[14] di Pierro, F., Khu, S-T. and Savic, D.A. 2007. An Investigation on Preference Order Ranking Scheme for Multiobjective Evolutionary Optimization. In *Proceedings of IEEE Transactions on Evolutionary Computation*. vol. 11, no. 1. 17–45.

[15] Fenton, N. and Pfleeger, S. L. 1997. *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. London, UK, International Thomson Computer Press.

[16] Ferrucci, F., Harman, M., Ren, J. and Sarro, F. 2013. Not going to take this anymore: multi-objective overtime planning for software engineering projects. In *Proceedings of the 2013 International Conference on Software Engineering* (ICSE '13). IEEE Press, Piscataway, NJ, USA, 462-471.

[17] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. 1999. *Refactoring – Improving the Design of Existing Code*. 1st ed. Addison-Wesley.

[18] Harman, M. 2013. Software Engineering: An Ideal Set of Challenges for Evolutionary Computation, In *Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion* (GECCO '13 Companion), Christian Blum (Ed.). ACM, New York, NY, USA, 1759-1760. DOI=10.1145/2464576.2480770.

[19] Harman, M. and Tratt, L. 2007. Pareto optimal search based refactoring at the design level. In *Proceedings of the Genetic and Evolutionary Computation Conference* (GECCO'07). 1106–1113.

[20] Harman, M., Mansouri, S. A. and Zhang, Y. 2012. Search-based software engineering: Trends, techniques and applications. In *ACM Computing Surveys*. vol. 45, no.1. Article 11 (December 2012), 61 pages. DOI=10.1145/2379776.2379787.

[21] Harman, M., and Jones, B., 2001. *Search Based Software Engineering*, Journal of Information and Software Technology, 43(14):833-839.

[22] Jaimes, A. L., Coello Coello, C.A. and Barrientos, J. E. U. 2009. Online Objective Reduction to Deal with Many-objective Problems. In *the 5th international conference on Evolutionary Multicriterion Optimization*. 423–437.

[23] Jain, H. and Deb, K. 2013. An Improved Adaptive Approach for Elitist Nondominated Sorting Genetic Algorithm for Many-Objective Optimization. In *Proceedings of Evolutionary Multicriterion Optimization* (EMO'13) 307–321.

[24] Jensen A. and Cheng B., 2010. On the use of genetic programming for automated refactoring and the introduction of design patterns. In Proceedings of GECCO. ACM.

[25] Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., and Ouni, A. 2011. Design Defects Detection and Correction by Example. In *Proceedings of the 19th IEEE International Conference on Program Comprehension* (ICPC'11). 81-90.

[26] Kilic, H., Koc, E. and Cereci, I. 2011. Search-based parallel refactoring using population-based direct approaches. In *Proceedings of the Third international Conference on Search Based Software Engineering* (SSBSE'11). 271–272.

[27] Kukkonen, S. and Lampinen, J. 2007. Ranking-Dominance and Many-Objective Optimization. In *Proceedings of IEEE Congress on Evolutionary Computation*. 3983-3990.

[28] Martínez, S. Z., Oropeza, E. G. Y. and Coello Coello, C. A. 2011. Self-adaptation techniques applied to multi-objective evolutionary algorithms. In *Proceedings of the 5th international conference on Learning and Intelligent Optimization* (LION'05). Carlos Coello Coello Ed.

[29] Moha, N. and Guéhéneuc, Y. 2007. Decor: a tool for the detection of design defects. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering* (ASE '07). ACM, New York, NY, USA, 527-528. DOI=10.1145/1321631.1321727.

[30] Moha, N., Guéhéneuc, Y.-G., Duchien, L. and Le Meur, A.-F. 2009. DECOR: A Method for the Specification and Detection of

Code and Design Smells. In *Proceedings of IEEE Transaction in Software Engineering* (TES'09). 20–36.

[31] Ó Cinnéide, M., Tratt, L., Harman, M., Counsell, S. and Moghadam, I. H. 2012. Experimental Assessment of Software Metrics Using Automated Refactoring. In *Proceedings of the Empirical Software Engineering and Management* (ESEM'12). ACM, New York, NY, USA, 49-58. DOI=10.1145/2372251.2372260.

[32] O'Keeffe, M. K. and Ó Cinnéide, M. 2008. Search-based refactoring for software maintenance. In *Journal of Systems and Software*. vol. 81, no.4. 502-516.

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

[34] Ouni, A., Kessentini, M., Sahraoui, H. and Boukadoum, M. 2012. Maintainability Defects Detection and Correction: A Multi-Objective Approach. *Journal of Automated Software Engineering*. Springer. vol. 20, no. 1, 47-79. DOI= 10.1007/s10515-011-0098-8.

[35] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A. and Poshyvanyk, D. 2013. Detecting Bad Smells in Source Code Using Change History Information. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (ASE 2013).

[36] Sato, H., Coello Coello, C. A., Aguirre, H. E. and Tanaka, K. 2012. Adaptive Control of the Number of Crossed Genes in Many-Objective Evolutionary Optimization. In *Proceedings of the 6th international conference on Learning and Intelligent Optimization* (LION'12), Youssef Hamadi and Marc Schoenauer (Eds.). Springer-Verlag, Berlin, Heidelberg, 478-484. DOI=10.1007/978-3-642-34413-8_48.

[37] Saxena, D.K., Duro, J.A., Tiwari, A., Deb, K. and Zhang, Q. 2013. Objective Reduction in Many-objective Optimization: Linear and Nonlinear Algorithms. In *Proceedings of IEEE Transactions on Evolutionary Computation*. vol. 17, no. 1. 77–99.

[38] Sayyad, A., Menzies, T. and Ammar, H. 2013. On the value of user preferences in search-based software engineering: a case study in software product lines. In *Proceedings of the 2013 International Conference on Software Engineering* (ICSE '13). IEEE Press, Piscataway, NJ, USA, 492-501.

[39] Sayyad, A.S., Ingram, J., Menzies, T. and Ammar, H. 2013 Scalable Product Line Configuration: A Straw to Break the Camel's Back. In *the International Conference on Automated Software Engineering* (ASE'13). Palo Alto, USA. November 2013. To Appear.

[40] Seng, O., Stammel, J. and Burkhart, D. 2006. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the Genetic and Evolutionary Computation Conference* (GECCO'06). 1909–1916.

[41] Singh, H. K., Isaacs, A. and Ray, T. 2011. A pareto corner search evolutionary algorithm and dimensionality reduction in many-objective optimization problems. In *Proceedings of IEEE Transactions on Evolutionary Computation*. vol. 99. 1–18.

[42] Talbi E. 2009. *Metaheuristics - From design to implementation*. John Wiley & Sons, ISBN: 978-0-470-27858-1.

[43] Thiele, L., Miettinen, K., Korhonen, P. J. and Luque, J. M. 2009. A Preference-Based Evolutionary Algorithm for Multi-Objective Optimization. In *Evolutionary Computation*. vol. 17, no. 3. 411-436.

[44] Wang, R., Purshouse, R. C. and Fleming P. J. 2013. Preference-Inspired Coevolutionary Algorithms for Many-Objective Optimization. In *Proceedings of IEEE Transaction in Evolutionary Computation* vol. 17, no. 4. 474-494.

[45] Yamashita A., and Moonen, L., 2012. Do code smells reflect important maintainability aspects?, Proceedings of the International Conference on Software Maintenance (ICSM).

[46] Zhang, Q. and Li, H. 2007. MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. In *Proceedings of IEEE Transactions on Evolutionary Computation*. vol. 11, no. 6. 712–731.

[47] Zitzler, E. and Künzli, S. 2004. Indicator-based selection in multiobjective search Parallel Problem Solving from Nature. In *Proceedings of the 8th International Conference on Parallel Problem Solving from Nature* (PPSN'04). Springer, Berlin, Germany. 832–842.

[48] (ApacheAnt). [Online]. Available: http://ant.apache.org/

[49] (ArgoUML). [Online]. Available: http://argouml.tigris.org/

[50] (Azureus). [Online]. Available: http://vuze.com

[51] (GanttProject). [Online]. Available: www.ganttproject.biz

[52] (Refactoring Browser). [Online]. Available: http://www.refactory.com/refactoring-browser

[53] (Refactorings in Alphabetical Order). [Online]. Available: http://www.refactoring.com/catalog/index.html

[54] (Xerces-J). [Online]. Available: http://xerces.apache.org/xerces-j/