Many-Objective Software Remodularization using NSGA-III

WIEM MKAOUER, University of Michigan
MAROUANE KESSENTINI, University of Michigan
PATRICE KONTCHEOU, University of Michigan
KALYANMOY DEB, Michigan State University
SLIM BECHIKH, University of Michigan
ALI OUNI, University of Michigan

Abstract. Systems nowadays are complex and difficult to maintain due to continuous changes and bad design choices. To handle the complexity of systems, software products are, in general, decomposed in terms of packages/modules regrouping classes that are dependent. However, it is challenging to automatically re-modularize systems to improve their maintainability. The majority of existing remodularization work mainly satisfy one objective which is improving the structure of packages by optimizing coupling and cohesion. In addition, most of existing studies are limited to only few operation types such as move class and split packages. Many other objectives are important to improve the automation of software re-modularization such as the design semantics, reducing the number of changes and maximizing the consistency with development change history. In this paper, we propose a novel many-objective search-based approach using NSGA-III to improve the automation of software remodularization. The process aims at finding the optimal remodularization solutions that improve the structure of packages, minimize the number of changes, preserve semantics coherence, and re-use the history of changes. We evaluate the efficiency of our approach using four different open-source systems and one automotive industry project, provided by our industrial partner, through a quantitative and qualitative study conducted with software engineers.

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

General Terms: Algorithms, Reliability.

Additional Key Words and Phrases: Search-based software engineering, software maintenance, software quality, re-modularization.

1. INTRODUCTION

Large scale software systems evolve and become quickly complex, fault-prone and difficult to maintain [34][6]. In fact, most of the changes during the evolution of systems such as introducing new features or fixing bugs are conducted, in general, within strict deadlines [50]. As a consequence, these code changes can have a negative impact on the quality of systems design such as the distribution of the classes in packages. To address this issue, one of the widely used techniques is software remodularization, called also software restructuring, which improves the existing decomposition of systems [1][2][10][13].

There has been much work on different techniques and tools for software remodularization [1][2][3][4][5][6][7][10][12]. The majority of existing studies focus more on the problem of clustering to find the best decomposition of a system in terms of modules rather than improving existing modularizations. In both categories, cohesion and coupling are the main metrics used to improve the quality of existing packages (e.g. modules) by determining which classes that should belong together in a package. In this paper, we focus on restructuring software design and not on the decomposition of systems to generate an initial coherent object oriented design.

Permission to make digital or hardcopies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with
The majority of existing contributions have formulated the restructuring problem as a single-objective problem where the goal is to improve the cohesion and coupling of packages [10][12][15][16]. Even though most of the existing approaches are powerful enough to provide remodularization solutions, some issues still need to be addressed. One of the most important issues is the semantic coherence of the design which is not considered by most of existing approaches. In fact, the definition of semantic is only limited to the cohesion measure without any consideration of syntactic measures (e.g. name of classes, name packages, etc.). Consequently, the restructured program could improve structural metrics but semantically incoherent. In this case, the design will become difficult to understand since classes are placed in wrong packages to improve the structure, cohesion and coupling. In addition, the number of code changes is not considered when suggesting remodularization solutions; the only aim is to improve the structure of packages independently of the cost of code changes. However, in a real-world scenario developers prefer, in general, remodularization solutions that improve the structure with a minimum number changes. It is important to minimize code changes to help developers in understanding the design after applying suggested changes. Furthermore, existing remodularization studies are limited to few types of changes mainly move class and split packages [10][11][12][13][16]. However, refactoring at the class and method levels can improve remodularization solutions such as by moving methods between classes located in different packages. In addition, most of existing work did not consider the software development history when suggesting changes to improve the structure of packages. The use of development history can be an efficient aid when proposing remodularization solutions. Packages that are modified over the past in the same period are, in general, semantically connected. Furthermore, packages that are extensively modified in the past have a high probability of being also changed in the future. Moreover, the packages to modify can be similar to some patterns that can be found in the development history, thus, developers can easily adapt them. Despite its importance, the history of code changes has not been widely investigated or used to suggest new remodularization solutions.

In this paper, we propose, for the first time, a many-objective search-based [35][27][70][71] to address the above-mentioned limitations. The process aims at finding the remodularization solution that: 1) Improve the structure of packages by optimizing some metrics such as number of classes per package, number of packages, coupling and cohesion; 2) Improve the semantic coherence of the restructured program. We combine two heuristics to estimate the semantic proximity between packages when moving elements between them (vocabulary similarity, and dependencies between classes extracted from call graphs) and some semantic/syntactic heuristics depending on the change type; 3) Minimize code changes. We consider new changes, comparing to existing remodularization studies, that can be related to the package, class and method levels such as extract package, move class, move method, etc.; and 4) Maximize the consistency with development change history. To better guide the search process, recorded code changes that are applied in the past in similar contexts are considered. We evaluate if similar changes are applied in previous versions of the packages that will be modified by the suggested remodularization solution.

The number of objectives to consider in our problem formulation is high (more than
three objectives); such problems are termed many-objective. In this context, the use of traditional multi-objective techniques, e.g., NSGA-II [29], widely used in search-based software engineering (SBSE), is clearly not sufficient like in our case for the problem of software remodularization. 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 [26][27][39][40][44] using e.g., objective reduction, new preference ordering relations, decomposition, etc. However, these techniques have not yet been widely explored in SBSE [37][35]. To the best of our knowledge and based on recent SBSE surveys [37][35], only one work exists proposed by [55][56] that uses a many-objective approach, IBEA (Indicator-Based Evolutionary Algorithm) [31], to address the problem of software product line creation. However, the number of considered objectives is limited to five.

We propose for the first time a scalable search-based software engineering approach based on NSGA-III [27] where there are 7 different objectives to optimize. Thus, in our approach, automated remodularization solutions will be evaluated using a set of 7 measures as described above. NSGA-III is a very recent many-objective algorithm proposed by Deb et al. [27]. The basic framework remains similar to the original NSGA-II algorithm [29], with significant changes in its selection mechanism. This paper represents one of the first real-world applications of NSGA-III and the first scalable work that supports the use of 7 objectives to address and improve the automation of software remodularization.

We evaluated our approach on four open source systems [66][69][70][71] and one industrial system provided by our industrial partner Ford Motor Company. We report the results on the efficiency and effectiveness of our approach, compared to the state of the art remodularization approaches [10][11][16]. Our results indicate that our approach significantly outperforms, in average, existing approaches in terms of improving the structure, reducing the number code changes, and semantics preservation.

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

1. The paper introduces a novel formulation of the remodularization problem as a many-objective problem that consider several objectives such as structural improvement, semantic coherence, number of changes and consistency with history of changes. It also the first work that consider the use of development history in the context software remodularization. This paper represents also the first real-world application of NSGA-III.

2. We consider in the paper the use of new operations, comparing to existing remodularization studies, such as move method, extract class, merge packages, etc.

3. The paper reports the results of an empirical study of our many-objective technique compared to different existing approaches [10][11][16]. The obtained results provide evidence to support the claim that our proposal is, in average, more efficient than existing techniques based on a benchmark of four large open source systems and one industrial project.

4. The qualitative evaluation of the results by software engineers at Ford Motor Company and also graduate students confirms the usefulness the suggested remodularization solutions.
The remainder of this paper is structured as follows: Section 2 is dedicated to the background needed to understand our approach and to the remodularization challenges. In Section 3, we describe how software remodularization is formulated as many-objective optimization problem and explain how we adapted the NSGA-III algorithm to our problem in Section 4. Section 5 presents and discusses the evaluation results on several medium and large size projects and we discuss the threats to validity related to our experiments. Related work is outlined in section 6. Section 7 concludes the paper and suggests future research directions.

2. SOFTWARE REMODULARIZATION: OPEN ISSUES

2.1 Challenges

Large systems such as automotive industry applications have to run and evolve over decades. As described by the law of Lehman and Belady [72], most of industrial systems must evolve and the design is, in general, extended far away the initial structure. Thus, it is mandatory to restructure the program design to reduce the cost of possible future evolutions. To this end, software remodularization is an important component in software maintenance activities.

Object oriented software modularization consist of regrouping a set of classes $C$ in terms of packages $P$. Thus, each package $P$ contains a set of classes. Several types of dependencies between packages can be found in the literature [1][2][9][11][13][15]. In this paper, we use the definition of dependencies between packages defined in [15]. Two main types of dependencies are described: 1- intra-edges dependencies and 2-inter-edges dependencies. The intra-edges include all types of intern dependencies between classes in the same package such as method call, class reference, inheritance, etc. The inter-edges include external dependencies between classes that are not in the same package. As illustrated in Figure 1, the system includes 2 packages, 3 intra-edges such as $(c_3, c_4)$ and 2 inter-edges such as $(c_1, c_3)$ for package $P_1$.

![Figure 1. The dependency graph including two packages, 3 intra-edges and 2 inter-edges](image)

Most of existing approaches are based on the use of cohesion and coupling to evaluate remodularization solutions [1][2][11][12][13][15][16]. The best solutions are those that maximize cohesion and minimize coupling. Cohesion of packages is, in general, defined by the number of intra-edges of all packages and coupling as the number of inter-edges of all packages. Even though most of the existing refactoring approaches are powerful enough to provide remodularization solutions, some open issues need to be targeted to provide an efficient and fully automated remodularization.
Packages structure improvements: Most of existing approaches consider remodularization as the process to improve the packages design quality by improving structural metrics. However, these metrics can be sometimes conflicting and it is difficult to find a compromise between them: moving classes to reduce the size or complexity of a package may increase the global coupling, increasing the cohesion of packages and minimizing coupling may reduce the number of packages and increase the number of classes in a package, etc. Thus, it is important to treat all these structural metrics as separate objectives to find the best trade-off between them. However, these structural metrics are not sufficient to generate efficient remodularization solutions.

Semantics domain: In object-oriented (OO) programs, objects reify domain concepts and/or physical objects. They implement their characteristics and behavior. Unlike other programming paradigms, grouping data and behavior into classes is not guided by development or maintenance considerations. Operations and fields of classes characterize the structure and behavior of the implemented domain elements. Consequently, a program could be syntactically correct, implement the right behavior, but violates the domain semantics if the reification of domain elements is incorrect. During the initial design/implementation, programs capture well the domain semantics when the OO principles are applied. However, when these programs are (semi) automatically modified/refactored during maintenance, the adequacy with domain semantics could be compromised. Semantics preservation is an important issue to consider when applying remodularization solutions.

Existing approaches suggest remodularization solutions mainly with the perspective of only improving some design/structural metrics such as distribution of classes in packages, coupling and cohesion. This, however, may not be sufficient enough. Indeed, a program could be syntactically correct, and have the right behavior, but it may model the domain semantics incorrectly which make the design difficult to understand by developers after remodularization. We need to preserve the rationale behind why and how classes are grouped and connected when applying change operations to improve the system modularization. More concretely, let us consider, for example, a code change that moves a class “ResearchLab” from the package “University” toward the package “Car”. This code change could improve the program modularization by reducing the number of classes in “University” and satisfies the pre- and post-conditions to preserve the behavior. However, having a class “Research” in package “Car” does not make sense from the domain semantics standpoint.

Code changes: When restructuring packages, different code changes are performed. The amount of code changes corresponds to the number of code elements (e.g., packages, classes, methods, fields, relationships, field references, etc.) modified through adding, deleting, or moving operations. Minimizing code changes when suggesting remodularization solutions is very important to help developers understand the modified/improved design. In fact, most developers want to keep as much as possible the existing design structure during the remodularization process [14]. Hence, improving the modularization of systems and reducing code changes are conflicting. Another interesting observation is that existing remodularization studies are limited to two types of code changes: moving classes between packages or just splitting packages [10][12][16]. However, the remodularization systems should evolve several other code changes such as merging packages, moving methods between classes (mainly between classes that are not in the same package), etc. For
example, it is sufficient to move a method between classes that are not in the same package to minimize coupling instead of moving the whole class.

Consistency with development/maintenance history: The majority of existing work did not consider the history of changes applied in the past when proposing new remodularization solutions. However, the history of code changes can be helpful in increasing the correctness of new restructuring solutions. To better guide the search process, recorded code changes applied in the past can be considered when proposing new changes such as move classes in similar contexts. This knowledge can be combined with structural and semantic information to improve the automation of software remodularization.

2.2 Motivating Example

To illustrate some of the above mentioned issues, Figure 2 shows a concrete example extracted from Xerces-J v2.7.0 [69], a well-known java open-source library for parsing, validating and manipulating XML documents. We consider a design fragment containing four packages \texttt{org.apache.xerces.validators.dtd}, \texttt{org.apache.xerces.dom}, \texttt{org.apache.xerces.domx}, and \texttt{org.apache.xerces.dom.events}. The largest package in Xerces-J v2.7.0 is \texttt{org.apache.xerces.dom} including a high number of large classes, comparing to all other packages, implementing several features in one package. Eleven software engineers out of the twelve that we asked in our experiments agreed that \texttt{org.apache.xerces.dom} is a large package that monopolizes the behavior of a large part of the system.

We consider an example of a remodularization solution that consists of moving the class \texttt{DeferredDocumentImpl} from the package \texttt{org.apache.xerces.dom} to the package \texttt{org.apache.xerces.validators.dtd}. This operation can improve the modularization quality by reducing the number of classes/functionalities of the package \texttt{org.apache.xerces.dom}. However, from the semantics coherence standpoint, this code change is incoherent since \texttt{DeferredDocumentImpl} provides the primary access to the XML document's data and not for the task of checking/validate the grammar structures (\texttt{org.apache.xerces.validators.dtd}). Based on semantic and structural information, using respectively a vocabulary-based similarity, and cohesion/coupling, many other target packages are possible including \texttt{org.apache.xerces.domx}, and \texttt{org.apache.xerces.dom.events}. These two packages have almost the same structure based on metrics such as number of classes and their semantic similarity is close to \texttt{org.apache.xerces.dom} using a vocabulary-based measure, or cohesion and coupling, etc. [73][74][75][76]. Thus, moving elements between these three packages is likely to be semantically meaningful. On the other hand, from previous versions of Xerces-J, we recorded that there are some classes (such as \texttt{AttributeMap}, \texttt{ParentNode}, \texttt{ChildNode}) that have been moved from the package \texttt{org.apache.xerces.domx} to the package \texttt{org.apache.xerces.dom}. As a consequence, moving classes from the package \texttt{org.apache.xerces.dom} to the package \texttt{org.apache.xerces.domx} has higher correctness probability than moving classes between the remaining packages.

Based on these observations, we believe that it is important to consider additional objectives rather than using only structural metrics to improve the automation of software remodularization. However, in most of the existing remodularization work, semantic coherence, code changes, and development history are not considered. Thus, the remodularization process needs a manual inspection by the user to evaluate the meaningfulness/feasibility of proposed changes that mainly improve structural metrics. The inspection aims at verifying if these changes could produce semantic
incoherence in the program design. For large-scale systems, this manual inspection is complex, time-consuming and error-prone. Improving the packages structure, minimizing semantic incoherencies, reducing code changes, and keeping consistent with development change history are conflicting. In some cases, improving the program modularization could provide a design that does not make sense semantically or could change radically the initial design. For this reasons, a good remodularization strategy needs to find a compromise between all of these objectives. In addition, moving classes and splitting packages are not enough code changes to improve the remodularization of systems. These observations are at the origin of the work described in this paper.

Figure 2. Motivating example extracted from Xerces-J v 2.7.0 [69]

3. SOFTWARE REMODULARIZATION : A MANY-OBJECTIVE PROBLEM

We describe in this section how many-objective techniques can be adapted to the software remodularization problem. We start first by describing many-objective techniques then we illustrate how the software remodularization problem can be considered as a many-objective one.

3.1 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 [26][27][31]. 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 [23]:

$$
\begin{align*}
\text{Min } f(x) &= [f_1(x), f_2(x), \ldots, f_M(x)]^T, \quad M > 3 \\
g_j(x) &\geq 0, \quad j = 1, \ldots, P; \\
h_k(x) &= 0, \quad k = 1, \ldots, Q; \\
x_i^L &\leq x_i \leq x_i^U, \quad i = 1, \ldots, n
\end{align*}
$$
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, $x_{iL}$ and $x_{iU}$ 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 $\Omega$.

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 [29], 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 [39]; 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 objective 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 [54]. 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 [48]. 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 [22], [23]. 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 [31], 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 [63]. Finally, we find the use of a predefined multiple targeted search. Inspired by preference-based MOEAs and the decomposition approach, recently, Deb and Jain [26], [40] and Wang et al. [61] 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 has demonstrated very promising results on MOPs involving up to 15 objectives.

Table 1. Summary of many-objective approaches.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Basic idea</th>
<th>Example algorithms</th>
<th># objectives</th>
<th>Applications</th>
</tr>
</thead>
</table>
| 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 [28]  
2) PCSEA [58] | 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) ε-NSGA-II [23]  
2) PBEA [60]  
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 [31]  
2) Ranking dominance-based algorithm [44]  
3) IBEA [64]  
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 [63] | 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) in the objective space. | 1) PICEA [61]  
2) NSGA-III [27] | 10 15 | 1) Crash-worthiness Design of Vehicles  
2) Nothing |
According to a recent survey by Harman et al. [35], 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 [55][56][51] 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 remodularization problem where up-to 7 objectives are considered to evaluate remodularization suggestions.

3.2 Many-Objective Software Remodularization: Overview

3.2.1 Approach Overview

Our approach aims at exploring a huge search space to find optimal remodularization solutions, i.e., a sequence of change operations (e.g. move class, extract package, move method, etc.), to restructure packages. The search space is determined not only by the number of possible change combinations, but also by the order in which they are applied. A heuristic-based optimization method is used to generate remodularization solutions. We have seven objectives to optimize: 1) minimize the number of classes per package; 2) minimize the number of packages; 3) maximize package cohesion; 4) minimize package coupling; 5) minimize the number of semantic errors by preserving the way classes are semantically grouped and connected together; 6) minimize code changes needed to apply remodularization solution; and 7) maximize the consistency with development change history. We consider the remodularization task as a many-objective optimization problem instead of a single-objective one using the new many-objective non-dominated sorting genetic algorithm (NSGA-III) that will be described in Section 4.

The general structure of our approach is sketched in Figure 3. It takes as input the source code of the program to be restructured, a list of possible remodularization operations (e.g. move class, extract package, move method, etc.) that can be applied, a set of semantic and structural measures, and a history of applied changes to previous versions of the system. Our approach generates as output the optimal sequence of operations, selected from an exhaustive list of possible ones that improve the structure of packages, minimize code changes needed to apply the remodularization solution, preserve the semantics coherence, and maximize the consistency with development change history. In the following, we describe the formal formulation of these different remodularization objectives to optimize.

3.2.2 Remodularization Objectives
We consider seven objectives to optimize in our many-objective adaptation to the software remodularization problem.

**Structure.** Four conflicting objectives are related to improving the structure of packages: 1) number of classes per package (minimize); 2) number of packages in the system (minimize); 3) cohesion to maximize that corresponds to the number of intra-edges (calls between classes in the same package) as described in Section 2; and 4) coupling to minimize that corresponds to the number of inter-edges (class between classes in different packages).

**Number of code changes.** Table 2 describes the types of Remodularization Operations (ROs) that are considered by our approach: Move method, Extract class, Move class, Merge packages, and Extract/Split package. Existing remodularization studies are limited to only two operation types: move class and split/extract package. We believe that these two operations are not enough to generate good remodularization solutions. In fact, sometimes only part of the class should be moved to another package (e.g. methods) and not the whole class. To apply a remodularization operation we need to specify which actors, i.e., code fragments, are involved in this refactoring and which roles they play when performing the change. As illustrated in Table 1, an actor can be a package, class, or method, statement and we specify for each operation the involved actors and their roles. It is important to minimize the number of suggested operations in the remodularization solution since the designer can have some preferences regarding the percentage of deviation with the initial program modularization. In addition, most of developers prefer solutions that minimize the number of changes applied to their design [17].

<table>
<thead>
<tr>
<th>Type of the operation</th>
<th>Actors</th>
<th>Roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move method</td>
<td>class</td>
<td>source class, target class</td>
</tr>
<tr>
<td></td>
<td>method</td>
<td>moved method</td>
</tr>
<tr>
<td>Extract class</td>
<td>class</td>
<td>source class, new class</td>
</tr>
<tr>
<td></td>
<td>method</td>
<td>moved methods</td>
</tr>
<tr>
<td>Move class</td>
<td>package</td>
<td>source package, target package</td>
</tr>
<tr>
<td></td>
<td>class</td>
<td>moved class</td>
</tr>
<tr>
<td>Merge packages</td>
<td>package</td>
<td>source package, target package</td>
</tr>
<tr>
<td>Extract/Split package</td>
<td>package</td>
<td>source package, target package</td>
</tr>
<tr>
<td></td>
<td>class</td>
<td>moved class</td>
</tr>
</tbody>
</table>

**Similarity with history of code changes.** The idea is to encourage the use of remodularization operations that are similar to those applied to the same code fragments (packages) in the past. To calculate the similarity score between a proposed operation and a recorded code change, we use the following function:

\[
\text{Sim}_\text{history}(RO) = \sum_{j=1}^{n} w_j
\]

where \( n \) is the number of recorded operations applied to the system in the past, and \( w_j \) is a change weight that reflects the similarity between the suggested remodularization operation (RO) and the recorded code change \( j \). The weight \( w_j \) is computed as follows: if the remodularization operations being compared are exactly the same (e.g., “Move Class” between the source and target same packages), the weight \( w_j = 2 \). If the operations being compared are similar (we consider two operations as similar if one of them is composed of the other or if their
implementations are similar, using equivalent controlling parameters (i.e., the same code fragments) as described in Table 1). Some complex operations, such as “Extract package”, can be composed of other operations such as “Move Method”, “Move class”, etc., the weight \( w_j = 1 \). Otherwise, \( w_j = 0 \).

**Semantics.** Most of remodularization operations are simple to implement and it is almost trivial to show that they preserve the behaviour [8]. However, until now there is no consensual way to investigate whether a code change can preserve semantic coherence of the original program/design. To preserve the semantics design, some constraints should be satisfied to ensure the correctness of the applied operations. We distinguish between two kinds of constraints: structural constraints and semantic constraints. Structural constraints were extensively investigated in the literature. Opdyke, for example, defined in [50] a set of pre and post-conditions for a large list of operations to ensure the structural consistence. Developers should check manually all actors related to the operation to inspect the semantic relationship between them.

We formulate semantics constraints using different measures in which we describe the concepts from a perspective that helps in automating the remodularization task:

**Vocabulary-based similarity (VS)**

This kind of similarity is interesting to consider when moving methods, or classes or merging packages or extract package. For example, when a class has to be moved from one package to another, the operation would make sense if both actors (source class and target packages) use similar vocabularies [73][74][75][76]. The vocabulary could be used as an indicator of the semantic similarity between different actors that are involved when performing a remodularization operation. We start from the assumption that the vocabulary of an actor is borrowed from the domain terminology and therefore could be used to determine which part of the domain semantics is encoded by an actor. Thus, two actors could be semantically similar if they use similar vocabularies.

The vocabulary could be extracted from the names of packages, classes, methods, fields, variables, parameters, types, etc. Tokenisation is performed using the Camel Case Splitter [75] which is one of the most used techniques in Software Maintenance tools for the preprocessing of identifiers. A more pertinent vocabulary can also be extracted from comments, commits information, and documentation. We calculate the semantic similarity between actors using information retrieval-based techniques (e.g., cosine similarity). The following equation calculates the cosine similarity between two actors. Each actor is represented as an n dimensional vector, where each dimension corresponds to a vocabulary term. The cosine of the angle between two vectors is considered as an indicator of similarity. Using cosine similarity, the conceptual similarity between two actors \( c_1 \) and \( c_2 \) is determined as follows:

\[
\text{Sim}(c_1, c_2) = \cos(c_1, c_2) = \frac{c_1 \cdot c_2}{\|c_1\| \cdot \|c_2\|} = \frac{\sum_{i=1}^{n} (w_{i,1} \cdot w_{i,2})}{\sqrt{\sum_{i=1}^{n} (w_{i,1})^2} \cdot \sqrt{\sum_{i=1}^{n} (w_{i,2})^2}} \in [0,1]
\]

where \( c_1 = (w_{i,1}, ..., w_{n,1}) \) is the term vector corresponding to actor \( c_1 \) and \( c_2 = (w_{i,2}, ..., w_{n,2}) \) is the term vector corresponding to \( c_2 \). The weights \( w_{i,j} \) can be computed using information retrieval based techniques such as the Term Frequency
Inverse Term Frequency (TF-IDF) method. We used a method similar to that described in [73] to determine the vocabulary and represent the actors as term vectors.

dependency-based similarity (DS)

We approximate domain semantics closeness between actors starting from their mutual dependencies. The intuition is that actors that are strongly connected (i.e., having dependency links) are semantically related. As a consequence, remodularization operations requiring semantic closeness between involved actors are likely to be successful when these actors are strongly connected. We consider two types of dependency links:

1) Shared method calls (SMC) that can be captured from call graphs derived from the whole program using CHA (Class Hierarchy Analysis)[76]. A call graph is a directed graph which represents the different calls (call in and call out) among all methods of the entire program. Nodes represent methods, and edges represent calls between these methods. CHA is a basic call graph which consider class hierarchy information, e.g, for a call c.m(...) assume that any m(...) is reachable that is declared in a subtype of the declared type of c. For a pair of actors, shared calls are captured through this graph by identifying shared neighbours of nodes related to each actor. We consider both, shared call-out and shared call-in. The following equations are used to measure respectively the shared call-out and the shared call-in between two actors $c_1$ and $c_2$ (two classes, for example). A shared method call is defined as the average of shared call-out and call-in.

\[
\text{sharedCallOut}(c_1, c_2) = \frac{|\text{callOut}(c_1) \cap \text{callOut}(c_2)|}{|\text{callOut}(c_1) \cup \text{callOut}(c_2)|} \in [0,1]
\]

\[
\text{sharedCallIn}(c_1, c_2) = \frac{|\text{callIn}(c_1) \cap \text{callIn}(c_2)|}{|\text{callIn}(c_1) \cup \text{callIn}(c_2)|} \in [0,1]
\]

2) Shared field access (SFA) that can be calculated by capturing all field references that occur using static analysis to identify dependencies based on field accesses (read or modify). We assume that two software elements are semantically related if they read or modify the same fields. The rate of shared fields (read or modified) between two actors $c_1$ and $c_2$ is calculated according to the following equation. In this equation, $\text{fieldRW}(c)$ computes the number of fields that may be read or modified by each method of the actor $c$. Thus, by applying a suitable static program analysis to the whole method body, all field references that occur could be easily computed.

\[
\text{sharedFieldsRW}(c_1, c_2) = \frac{|\text{fieldRW}(c_1) \cap \text{fieldRW}(c_2)|}{|\text{fieldRW}(c_1) \cup \text{fieldRW}(c_2)|} \in [0,1]
\]

cohesion-based dependency (CD)

The cohesion-based similarity that we propose for software remodularization is mainly used by the extract class, merge packages and extract package operations, is defined to find a cohesive set of classes, methods and attributes to be moved to the newly extracted class or package. A new class or package can be extracted from a source class or package by moving a set of strongly related (cohesive) classes, fields and methods from the original class or package to the new class or package. Extracting this set will improve the cohesion of the original package or class and minimize the coupling with the new package/class. Applying the “Extract Package/Class” or “Merge Packages” operation on a specific package/class will result...
in this class being split (or merged) into classes/packages. We need to calculate the semantic similarity between the elements in the original package/class to decide how to split or merge the original packages/classes.

We use vocabulary-based similarity and dependency-based similarity to find the cohesive set of actors (packages, classes, methods and fields). Consider a source package that contains $n$ classes $\{c_1, ..., c_n\}$, $m$ methods $\{m_1, ..., m_m\}$, and $p$ fields $\{f_1, ..., f_p\}$. We calculate the similarity between each pair of elements in a cohesion matrix. The cohesion matrix is obtained as follows: for the method-method or package-package similarity, we consider both vocabulary and dependency-based similarity. For the method-field similarity, if the method $m_i$ may access (read or write) the field $f_j$, then the similarity value is 1. Otherwise, the similarity value is 0. The column “Average” contains the average of similarity values for each line. The suitable set of methods and fields to be moved to a new class is obtained as follows: we consider the line with the highest average value and construct a set that consists of the elements in this line that have a similarity value that is higher than a certain threshold.

To find a compromise between the seven objectives described in this section, we used a recent many-objective optimization algorithm (NSGA-III) that will be described in the next section.

4. SOFTWARE REMODULARIZATION USING NSGA-III

This section shows how the remodularization problem can be addressed using NSGA-III. We first present an overview of NSGA-III then we provide the details of our adaptation to the remodularization problem.

4.1 NSGA-III

NSGA-III is a very recent many-objective algorithm proposed by Deb et al. [27]. The basic framework remains similar to the original NSGA-II algorithm [29] with significant changes in its selection mechanism. Figure 4 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 5). 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 $\rho$ 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 \( F_t \) 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 \( \rho_j \) for the \( j \)-th reference point. We now devise a new niche-preserving operation as follows. First, we identify the reference point set \( J_{\text{min}} = \{ j : \text{argmin}_i (\rho_i) \} \) having minimum \( \rho_j \). In case of multiple such reference points, one \( (j^* \in J_{\text{min}}) \) 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 \( F_t \) 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 \( \rho_{j^*} \) is then incremented by one. Second, the front \( F_t \) 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 \( F_t \) that is associated with the reference point \( F_t \) is added to \( P_{t+1} \). If such a member exists, the count \( \rho_{j^*} \) is incremented by one. After \( \rho_j \) 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 \( Z \), parent population \( P_t \)

**Output:** \( P_{t+1} \)

```
00: Begin
01: \( S_t \leftarrow \emptyset \), \( i \leftarrow 1 \);
02: \( Q_t \leftarrow \text{Variation} (P_t) \);
03: \( R_t \leftarrow P_t \cup Q_t \);
04: \( (F_t, F_{t+},..., ) \leftarrow \text{Non-dominationed_Sort} (R_t) \);
05: Repeat
06: \( S_t \leftarrow S_t \cup F_t \); \( i \leftarrow i+1 \);
07: Until \( |S_t| \geq N \);
08: \( F_t \leftarrow F_{t+} ; /*Last front to be included*/
09: If \( |S_t| = N \) then
10: \( P_{t+1} \leftarrow S_t \);
11: Else
12: \( P_{t+1} \leftarrow \bigcup_{j=1}^{i-1} F_{j}^t \);
   /*Number of points to be chosen from \( F_{j}^t \)*/
13: \( K \leftarrow N - |P_{t+1}| \);
   /*Normalize objectives and create reference set \( Z^* \)*/
14: \( \text{Normalize} (F^t; S_t, Z^*; Z^*) \);
   /*Associate each member \( s \) of \( S_t \) with a reference point*/
   /*\( \pi(s) \): closest reference point*/
   /*\( d(s) \): distance between \( s \) and \( \pi(s) \)*/
15: \( [\pi(s), d(s)] \leftarrow \text{Associate} (S_t, Z^*) \);
   /*Compute niche count of reference point \( j \in Z^* \)*/
16: \( \rho_j \leftarrow \sum_{s \in S_t/F_t} ((\pi(s) = j) \ ? 1 : 0) \);
   /*Choose \( K \) members one at a time from \( F_t \) to construct \( P_{t+1} \)*/
17: \( \text{Niching} (K, \rho_j, \pi(s), d(s), Z^*, F_t, P_{t+1}) \);
18: End If
19: End
```

**Figure 4. Pseudocode of NSGA-III main procedure.**
4.2 Solution Approach

This section describes our adaptation of NSGA-III to our remodularization problem. Thus, we define the following adaptation steps: representation of the solutions and the generation of the initial population, evaluation of individuals using the fitness functions, selection of the individuals from one generation to another, generation of new individuals using genetic operators (crossover and mutation) to explore the search space and the normalization of population members.

4.2.1 Solution Representation

To represent a candidate remodularization solution (individual), we used a vector representation. Each vector's dimension represents a remodularization operation. Thus, a solution is defined as a long sequence of operations applied to different parts of the system to improve its modularization. When created, the order of applying these code changes corresponds to their positions in the vector. In addition, for each operation, a set of controlling parameters, e.g., actors and roles, as illustrated in Table 2, are randomly picked from the program to be restructured. An example of a solution is given in Figure 6 applied to the motivating example described in Section 2.

![Normalized reference plane for a three-objective case](image)

Figure 5. Normalized reference plane for a three-objective case [26].

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move Class(AttrNSImpl, org.apache.xerces.dom, org.apache.xerces.validators.dtd)</td>
</tr>
<tr>
<td>Extract Class(XGrammarWriter, XGrammarInput, parseInt())</td>
</tr>
<tr>
<td>Move Method(normalize(), XGrammarWriter, DTDGrammar)</td>
</tr>
<tr>
<td>Extract Package(org.apache.xerces.dom, org.apache.xerces.dtd, CharacterDataImpl, ChildNode)</td>
</tr>
</tbody>
</table>

Figure 6. NSGA-III solution representation

When generating a sequence of operations (individual), it is important to guarantee that they are feasible and that they can be applied. The first work in the literature was proposed by Opdyke [50] who introduced a way of formalizing the preconditions that must be imposed before a code change can be applied in order to preserve the behavior of the system. Opdyke created functions which could be used to formalise constraints. These constraints are similar to the Analysis Functions used later by Ó Cinneide [48][49] who developed a tool to reduce program analysis. In our approach, we used a system to check a set of simple conditions, taking inspiration from the work proposed by Ó Cinnéide [49]. Our search-based refactoring tool simulates refactorings using pre and post conditions that are expressed in terms of conditions on a code model. For example, to apply the remodularization operation Move Class(AttrNSImpl, org.apache.xerces.dom, org.apache.xerces.validators.dtd), a number of necessary preconditions should be satisfied, e.g., org.apache.xerces.dom
and \texttt{org.apache.xerces.validators.dtd} should exist and should be packages; \texttt{AttrNSImpl} should exist and should be a class; the class \texttt{AttrNSImpl} should be implemented in the package \texttt{org.apache.xerces.dom}; etc. As postconditions, \texttt{AttrNSImpl}, and \texttt{org.apache.xerces.dom}, \texttt{org.apache.xerces.validators.dtd} should exist; \texttt{AttrNSImpl} class should be in the package \texttt{org.apache.xerces.validators.dtd} should not exist anymore in the package \texttt{org.apache.xerces.dom}.

4.2.2 Fitness Functions

Each generated remodularization solution is executed on the system S. Once all required data is computed, the solution is evaluated based on the 7 objectives described in Section 3. Based on these values, the remodulairzation 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. As a reminder, the following fitness functions are used: 1) number of classes per package (minimize); 2) number of packages in the system (minimize); 3) cohesion (maximize); 4) coupling (minimize); 5) Semantics coherence (maximize); 6) number of operations (minimize); and 7) coherence with the history of code changes (minimize). For the semantic fitness function that corresponds to the weighted sum of different semantic measures described in Section 3. The semantic fitness function of a solution corresponds to the average of the semantic values of the operations in the vector. For the history of changes fitness function that maximizes the use of operations that are similar to those applied to the same code fragments in the past. To calculate the similarity score between a proposed remodularization operation and a recorded operation, we use the fitness function described in section 3. Some code changes contribute to the domain vocabulary of the system but not all of them. Thus, we are considering semantic coherence and consistency with prior code changes as separate objectives. In addition, the consistency with priori code changes is not mainly related to the domain vocabulary but to the type of operations that are applied to similar context. Treating the similarity with prior changes as a separate objective can address the problem that developers use sometimes names of code elements that do not make any sense. Furthermore, we decided to separate semantic coherence and the consistency with prior code changes in two different objective to give more flexibility of the user to select the best solution based on its preferences and also to ensure the usefulness of our approach even in the situation when the change history is not available (the user can easily exclude this objective while maintaining the semantic coherence one).

Normalization of population members. Usually objective functions are incommensurable (i.e., they have different scales). For this reason, we used the normalization procedure proposed by Deb et al. [27] 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 prerequisite for diversity preservation.

4.2.3 Evolutionary Operators

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. As illustrated in Figure 7, each child combines some of the refactoring operations of the first parent with some ones of the second parent. In any
given generation, each solution will be the parent in at most one crossover operation. It is important to note that in many-objective optimization, it is better to create children that are close to their parents in order to have a more efficient search process [27][28]. 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 operations sequence or belonging to the last tier.

For mutation, we use the bit-string mutation operator that picks probabilistically one or more operations from its or their associated sequence and replace them by other ones from the initial list possible operations as described in the running example of Figure 8.

After applying genetic operators (mutation and crossover), we verify the feasibility of the generated sequence of operations by checking the pre and post conditions. Each operation that is not feasible due to unsatisfied preconditions will be removed from the generated refactoring sequence. The new sequence is considered valid in our NSGA-III adaptation if the number of rejected operations is less than 10% of the total sequence size.

5. VALIDATION

In order to evaluate our approach for restructuring systems using NSGA-III, we conducted a set of experiments based on different versions of large open source systems [70][71][69][66] and one industrial project provided by Ford Motor Company. 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 [10][11][16]. 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.

5.1 Research Questions

In our study, we assess the performance of our remodularization approach by finding out whether it could generate meaningful sequences of operations that
improve the structure of packages while reducing the number of code changes, preserving the semantic coherence of the design, and reusing as much as possible a base of recorded operations applied in the past in similar contexts. Our study aims at addressing the following research questions outlined below. We also explain how our experiments are designed to address these questions. The main question to answer is to what extent can the proposed approach proposes meaningful remodularization solutions. To find an answer, we defined the following 7 research questions:

RQ1.1: To what extent can the proposed approach improves the structure of packages in the system?

RQ1.2: To what extent the proposed approach preserves the semantics while improving the packages structure?

RQ1.3: To what extent can the proposed approach minimizes the number of changes (size)?

RQ1.4: To what extent the use of recorded changes improves the suggestion of good remodularization solutions?

RQ2: How does the proposed many-objective approach based on NSGA-III perform compared to other many/multi-objective algorithms or a mono-objective approach?

RQ3: How does the proposed many-objective approach based on NSGA-III perform compared to existing remodularization approach not based on heuristic search?

RQ4: Insight. How our many-objective re-modularization approach can be useful for software engineers in real-world setting?

To answer RQ1.1, we validate the proposed remodularization on four medium and large-size open-source systems [66][69][70][71] and one industrial project to evaluate the structural improvements of systems after applying the best solution. To this end, we used the following metrics: average number of classes per package (NCP), number of packages (NP), number of inter-edges (NIE) and number of intra-edges (NAE).

To answer RQ1.2, it is important to validate the proposed remodularization solutions from both quantitative and qualitative perspectives. To this end, we use two different validation methods: manual validation and automatic validation of the efficiency of the proposed solutions. For the manual validation, we asked groups of potential users (software engineers) of our remodularization tool to evaluate, manually, whether the suggested operations are feasible and make sense semantically. We define the metric “manual precision” (MP) which corresponds to the number of meaningful operations, in terms of semantic coherence, over the total number of suggested operations. MP is given by the following equation

$$MP = \frac{\#\text{coherent operations}}{\#\text{proposed operations}} \in [0,1]$$

For the automatic validation, we introduce manually several changes on the remodularization of JHotDraw and we evaluate the ability of our approach to generate the initial version of the system (considered as a well-designed system). In fact, JHotDraw is considered as one of the well-designed open source systems and several design patterns are used in his implementation. Thus, we compare the proposed operations with the expected ones in terms of recall and precision:

$$RE_{\text{recall}} = \frac{|\text{suggested operations} \cap |\text{expected operations}|}{|\text{expected operations}|} \in [0,1]$$

$$PR_{\text{precision}} = \frac{|\text{suggested operations} \cap |\text{expected operations}|}{|\text{suggested operations}|} \in [0,1]$$
To answer RQ1.3, we evaluate the number of operations (NO) suggested by the best remodularization solutions on the different systems.

To answer RQ1.4, we use the metric $MP$ to evaluate the effect of the use of recorded operations, applied in the past to similar contexts, on the semantic coherence. Moreover, in order to evaluate the importance of reusing recorded operations in similar contexts, we define the metric “reused operations” (ROP) that calculates the percentage of operations from the base of recorded operations used to generate the optimal remodularization solutions by our proposal. $ROP$ is given by the following equation

$$ROP = \frac{\# \text{ used operations from the base of recorded operations}}{\# \text{ operations in the base of refactorings}} \in [0,1]$$

To answer RQ2, we compared the performance of NSGA-III with two many-objective techniques, MOEA/D [63] and IBEA [64], and also with a multi-objective algorithm that uses NSGA-II [25]. We used Inverted Generational Distance (IGD) to compare between the different algorithms: 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 [27]. 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. In addition to IGD, we used the above described metrics to compare between all the algorithms: $NCP$, $NP$, $NIE$, $NAE$, $MP$, $RE$, and $PR$. We also compared our approach with a multi-objective remodularization technique proposed by Abdeen et al. [11] where the objectives considered are coupling, cohesion and number of changes. Since the approach of Abdeen et al. [10][11] is limited to the use of only one operation (Move Class), we only used the qualitative evaluation based on $MP$ for the comparison.

It is important also to determine if considering each conflicting metric as a separate objective to optimize performs better than a mono-objective approach that aggregates several 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 Knee point [23] (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. We compared NSGA-III with an existing mono-objective remodularization approach [10] based on the use of cohesion and coupling aggregated in one fitness function. Since the mono-objective approach [10] is limited to the use of only one operation (Move Class), we only used the qualitative evaluation based on $MP$ for the comparison and feed-backs from software engineers on using both tools.

For RQ3, since it is not sufficient to outperform existing search-based remodularization techniques, we compared our proposal to an existing remodularization technique based on the use of coupling and cohesion [16] and limited to the only use of “Split packages” change. Thus, we compared our proposal using only the qualitative evaluation based on $MP$ and feed-backs from software engineers on using both tools.
For RQ4, we evaluated the benefits of using our remodularization tool by several software engineers. To this end, they classify the suggested operations (IOP) one by one as interesting or not. The difference with the MP metric is that the operations are not classified from a semantic coherence perspective but form a usefulness one.

\[ IOP = \frac{\text{# useful operations}}{\text{# operations}} \in [0, 1] \]

### 5.2 Software Projects Studied

We used a set of well-known open-source java projects and one project from our industrial partner Ford Motor Company. We applied our approach to four large and medium size open-source java projects: Xerces-J [69], JFreeChart [71], GanttProject [66], and JHotDraw [70]. Xerces-J is a family of software packages for parsing XML. JFreeChart is a powerful and flexible Java library for generating charts. GanttProject is a cross-platform tool for project scheduling. JHotDraw is a GUI framework for drawing editors. Finally, the industrial project, JDI, is Java-based software system that helps Ford Motor Company analyze useful information from the past sales of dealerships data and suggests which vehicles to order for their dealer inventories in the future. This system is main key software application used by Ford Motor Company to improve their vehicles sales by selecting the right vehicle configuration to the expectations of customers. JDI is a highly structured and several versions were proposed by software engineers at Ford during the past 10 years. Due to the importance of the application and the high number of updates performed during a period of 10 years, it is critical to ensure good modularization of JDI to reduce the time required by developers to introduce new features in the future.

We selected these systems for our validation because they range from medium to large-sized open-source projects, which have been actively developed over the past 10 years, and their design has not been responsible for a slowdown of their developments. Table 3 provides some descriptive statistics about these six programs.

<table>
<thead>
<tr>
<th>Systems</th>
<th>Release</th>
<th># classes</th>
<th>KLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xerces-J</td>
<td>v2.7.0</td>
<td>991</td>
<td>240</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>v6.1</td>
<td>585</td>
<td>21</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>v1.0.9</td>
<td>521</td>
<td>170</td>
</tr>
<tr>
<td>GanttProject</td>
<td>v1.10.2</td>
<td>245</td>
<td>41</td>
</tr>
<tr>
<td>JDI-Ford</td>
<td>v5.8</td>
<td>638</td>
<td>247</td>
</tr>
</tbody>
</table>

To collect operations applied in previous program versions, we use Ref-Finder [30]. Ref-Finder, implemented as an Eclipse plug-in, can identify refactoring operations between two releases of a software system. Table 4 shows the analyzed versions and the number of operations, identified by Ref-Finder, between each subsequent couple of analyzed versions, after the manual validation. In our study, we consider only the operation types described in Table 1.

<table>
<thead>
<tr>
<th>Systems</th>
<th>Collected operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xerces-J</td>
<td>v1.4.2 - v2.6.1</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>v1.0.6 - v1.0.8</td>
</tr>
<tr>
<td>GanttProject</td>
<td>v1.7 - v1.10.1</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>v5.1 - v6.0</td>
</tr>
<tr>
<td>JDI-Ford</td>
<td>v2.4 - v5.6</td>
</tr>
</tbody>
</table>
5.3 Experimental Setting

The goal of the study is to evaluate the usefulness and the effectiveness of our remodularization tool in practice. We conducted a non-subjective evaluation with potential developers who can use our tool. Indeed, operations should not only improve the structure of packages, but should also be meaningful from a developer’s point of view in terms of semantic coherence and usefulness.

5.3.1 Subjects

Our study involved 13 subjects from the University of Michigan and 2 software engineers from Ford Motor Company. Subjects include 5 master students in Software Engineering, 7 PhD students in Software Engineering, 1 faculty member in Software Engineering, and 2 junior software developers. 4 of them are female and 11 are male. All the subjects are volunteers and familiar with java development. The experience of these subjects on Java programming ranged from 2 to 16 years.

5.3.2 Scenario

We designed our study to answer our research questions. The subjects were invited to fill a questionnaire that aims to evaluate our suggested refactorings. We divided the subjects into five groups according to 1) the number of studied systems (five systems of Table 4), 2) the number of remodularization solutions to evaluate, and 3) the number of techniques to be tested.

The number of remodularization solutions to evaluate depends on different objectives combinations: average number of classes per package (NCP), number of packages (NP), coupling (COU), cohesion (COH), semantics preservation (SP), number of changes (NCH) and coherence with history of changes (CHC). For each combination (two, three, four, five, six, seven, objectives), a remodularization solution is suggested to find the best compromise between the considered objectives. Similarly, the solutions of the state-of-the art works [10][11][16] are empirically evaluated in order to compare them to our approach as described in the previous section. Table 5 describes the number of remodularization solutions to be evaluated for each studied system in order to answer our research questions.

Table 5. Considered solutions for the qualitative evaluation

<table>
<thead>
<tr>
<th>Ref. Solution</th>
<th>Algorithm/ Approach</th>
<th># Objectives</th>
<th>Objectives considered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution 1</td>
<td>NSGA-III</td>
<td>2</td>
<td>NCP, NP</td>
</tr>
<tr>
<td>Solution 2</td>
<td>NSGA-III</td>
<td>3</td>
<td>NCP, NP, COU</td>
</tr>
<tr>
<td>Solution 3</td>
<td>NSGA-III</td>
<td>4</td>
<td>NCP, NP, COU, COH</td>
</tr>
<tr>
<td>Solution 4</td>
<td>NSGA-III</td>
<td>5</td>
<td>NCP, NP, COU, COH, SP</td>
</tr>
<tr>
<td>Solution 5</td>
<td>NSGA-III</td>
<td>6</td>
<td>NCP, NP, COU, COH, SP, NCH</td>
</tr>
<tr>
<td>Solution 6</td>
<td>NSGA-III</td>
<td>7</td>
<td>NCP, NP, COU, COH, SP, NCH, CHC</td>
</tr>
<tr>
<td>Solution 7</td>
<td>IBEA</td>
<td>2</td>
<td>NCP, NP</td>
</tr>
<tr>
<td>Solution 8</td>
<td>IBEA</td>
<td>3</td>
<td>NCP, NP, COU</td>
</tr>
<tr>
<td>Solution 9</td>
<td>IBEA</td>
<td>4</td>
<td>NCP, NP, COU, COH</td>
</tr>
<tr>
<td>Solution 10</td>
<td>IBEA</td>
<td>5</td>
<td>NCP, NP, COU, COH, SP</td>
</tr>
<tr>
<td>Solution 11</td>
<td>IBEA</td>
<td>6</td>
<td>NCP, NP, COU, COH, SP, NCH</td>
</tr>
<tr>
<td>Solution 12</td>
<td>IBEA</td>
<td>7</td>
<td>NCP, NP, COU, COH, SP, NCH, CHC</td>
</tr>
<tr>
<td>Solution 13</td>
<td>MOEA/D</td>
<td>2</td>
<td>NCP, NP</td>
</tr>
<tr>
<td>Solution 14</td>
<td>MOEA/D</td>
<td>3</td>
<td>NCP, NP, COU</td>
</tr>
<tr>
<td>Solution 15</td>
<td>MOEA/D</td>
<td>4</td>
<td>NCP, NP, COU, COH</td>
</tr>
<tr>
<td>Solution 16</td>
<td>MOEA/D</td>
<td>5</td>
<td>NCP, NP, COU, COH, SP</td>
</tr>
<tr>
<td>Solution 17</td>
<td>MOEA/D</td>
<td>6</td>
<td>NCP, NP, COU, COH, SP, NCH</td>
</tr>
<tr>
<td>Solution 18</td>
<td>MOEA/D</td>
<td>7</td>
<td>NCP, NP, COU, COH, SP, NCH, CHC</td>
</tr>
<tr>
<td>Solution 19</td>
<td>Abdeen et al.2011 [10] (mono-objective Simulate Annealing)</td>
<td>1</td>
<td>COU+COH</td>
</tr>
<tr>
<td>Solution 21</td>
<td>Bavota et al. 2013 [16] (not SBSE)</td>
<td>2</td>
<td>COU, COH</td>
</tr>
</tbody>
</table>
As shown in Table 5, for each system, 21 remodularization solutions have to be evaluated. Due to the huge number of operations to be evaluated (each solution consists of a set of operations), we pick at random a sub-set of up-to 10 operations per solution to be evaluated in our study. In Table 6, we summarize how we divided subjects into 6 groups in order to cover all remodularization solutions. In addition, as illustrated in Table 6, we are using a cross-validation to reduce the impact of subjects on the evaluation. Each subject evaluates different remodularization solutions for three different systems.

Subjects were aware that they are going to evaluate the semantic coherence and the usefulness of the operations, but do not know the particular experiment research questions (algorithms used, different objectives used and their combinations). Consequently, each group of subjects who accepted to participate to the study, received a questionnaire, a manuscript guide to help them to fill the questionnaire, and the source code of the studied systems, in order to evaluate 21 solutions (10 operations per solution). The questionnaire is organized in an excel file with hyperlinks to visualize easily the source code of the affected code elements. Subjects are invited to select for each refactoring operation one of the possibilities: "Yes", "No", or "May be" (if not sure) about the semantic coherence and usefulness. Since the application of remodularization solutions is a subjective process, it is normal that not all the programmers have the same opinion. In our case, we considered the majority of votes to determine if suggested solutions are correct or not.

<table>
<thead>
<tr>
<th>Subject groups</th>
<th>Systems</th>
<th>Algorithm / Approach</th>
<th>Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group A</td>
<td>GanttProject</td>
<td>NSGA-III IBEA</td>
<td>Solution 1-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 7-12</td>
</tr>
<tr>
<td></td>
<td>Xerces</td>
<td>MOEA/D, Abdeen et al.2011</td>
<td>Solution 13-18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 19</td>
</tr>
<tr>
<td></td>
<td>JFreeChart</td>
<td>Abdeen et al. 2013, Bavota et al. 2013</td>
<td>Solution 20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 21</td>
</tr>
<tr>
<td>Group B</td>
<td>GanttProject</td>
<td>NSGA-III IBEA</td>
<td>Solution 1-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 7-12</td>
</tr>
<tr>
<td></td>
<td>Xerces</td>
<td>MOEA/D, Abdeen et al.2011</td>
<td>Solution 13-18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 19</td>
</tr>
<tr>
<td></td>
<td>JFreeChart</td>
<td>Abdeen et al. 2013, Bavota et al. 2013</td>
<td>Solution 20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 21</td>
</tr>
<tr>
<td>Group C</td>
<td>GanttProject</td>
<td>NSGA-III IBEA</td>
<td>Solution 1-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 7-12</td>
</tr>
<tr>
<td></td>
<td>Xerces</td>
<td>MOEA/D, Abdeen et al.2011</td>
<td>Solution 13-18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 19</td>
</tr>
<tr>
<td></td>
<td>JFreeChart</td>
<td>Abdeen et al. 2013, Bavota et al. 2013</td>
<td>Solution 20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 21</td>
</tr>
<tr>
<td>Group D</td>
<td>GanttProject</td>
<td>NSGA-III IBEA</td>
<td>Solution 1-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 7-12</td>
</tr>
<tr>
<td></td>
<td>JHotDraw</td>
<td>MOEA/D, Abdeen et al.2011</td>
<td>Solution 13-18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 19</td>
</tr>
<tr>
<td></td>
<td>JDI-Ford</td>
<td>Abdeen et al. 2013, Bavota et al. 2013</td>
<td>Solution 20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 21</td>
</tr>
<tr>
<td>Group E</td>
<td>Xerces</td>
<td>NSGA-III IBEA</td>
<td>Solution 1-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 7-12</td>
</tr>
<tr>
<td></td>
<td>JHotDraw</td>
<td>MOEA/D, Abdeen et al.2011</td>
<td>Solution 13-18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 19</td>
</tr>
<tr>
<td></td>
<td>JDI-Ford</td>
<td>Abdeen et al. 2013, Bavota et al. 2013</td>
<td>Solution 20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 21</td>
</tr>
<tr>
<td>Group F</td>
<td>JFreeChart</td>
<td>NSGA-III IBEA</td>
<td>Solution 1-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 7-12</td>
</tr>
<tr>
<td></td>
<td>JHotDraw</td>
<td>MOEA/D, Abdeen et al.2011</td>
<td>Solution 13-18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Solution 19</td>
</tr>
<tr>
<td></td>
<td>JDI-Ford</td>
<td>Abdeen et al. 2013,</td>
<td>Solution 20</td>
</tr>
</tbody>
</table>

Table 6. Survey organization
5.3.3 Parameters Tuning

Parameter setting influences significantly the performance of a search algorithm on a particular problem [18]. For this reason, for search-based algorithm and for each system (cf. Table 7), we perform a set of experiments using several population sizes: 62, 100, 150, 180, 140, and 190 for respectively 2, 3, 4, 5, 6 and 7 objectives. The maximum number of generations used is 300, 500, 700, 1000, 1200, and 1400 respectively for 2, 3, 4, 5, 6 and 7 objectives. For each algorithm, to generate an initial population, we start by defining the maximum vector length (maximum number of operations per solution). The vector length is proportional to the number of operations that are considered and the size of the program to be restructured. A higher number of operations in a solution do not necessarily mean that the results will be better. Ideally, a small number of operations should be sufficient to provide a good trade-off between the fitness functions. This parameter can be specified by the user or derived randomly from the sizes of the program and the used operations list. During the creation, the solutions have random sizes inside the allowed range. 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 [44] 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 7 depicts the important common parameters.

<table>
<thead>
<tr>
<th>Number of objectives</th>
<th>Number of reference points (for NSGA-III and MOEA/D)</th>
<th>Population size</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>62</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>150</td>
</tr>
<tr>
<td>4</td>
<td>150</td>
<td>182</td>
</tr>
<tr>
<td>5</td>
<td>180</td>
<td>174</td>
</tr>
<tr>
<td>6</td>
<td>140</td>
<td>186</td>
</tr>
<tr>
<td>7</td>
<td>190</td>
<td>193</td>
</tr>
</tbody>
</table>

5.4 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 [18][19] with a 95% confidence level (α = 5%). The latter verifies the null hypothesis H0 that the obtained results of two algorithms are samples from continuous distributions with equal medians, against the alternative that they are not H1. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H0 while it is true (type I error). A p-value that is less than or equal to α (≤ 0.05) means that we accept H1 and we reject H0. 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.

5.5 Results

5.5.1 Results for RQ1.1
As described in Table 8, after applying the proposed operations (best remodularization solutions), we found that NSGA-III algorithm provides similar structural improvements the other techniques in terms of average number of classes per package (NCP), cohesion (NIE) and coupling (NAE). However, the number of packages (NP) in the system after applying NSGA-III solutions is higher than all NP values proposed by the best solutions of the remaining algorithms in most of the cases. This is can be explained by the fact that decreasing the number of classes per package will automatically increase the number of packages. An interesting observation that the difference between number of packages proposed by NSGA-III solutions is reasonable and the average difference in terms of number of packages is around only 4. The feedback that we received from the software engineers confirm that the number of packages is less important than the number of classes per package thus we can consider that overall NSGA-III modularization solutions improves, in average, the structures of systems better than all the other approaches.

The structural improvement scores of multi-objective and mono-objective algorithms are very close to those produced by many-objective algorithms especially NSGA-III. This is an interesting result confirming that our NSGA-III can found very good compromises between 7 objectives that are similar and sometimes outperforms those that are produced by existing approaches using only structural and semantic objectives.

We believe that improving the structure of packages it is a difficult and very important objective to reach. We consider that NSGA-III performance in terms of improving the structure similar to existing approaches is a very interesting result since the main goal of this work to improve the structure while preserving the domain semantics which not well-considered by the remaining approaches [10][11][16].

Table 8. Average number of classes per package (NCP), number of packages (NP), number of inter-edges (NIE), number of intra-edges (NAE) and the deviation (delta with the initial design) median values of NSGA-III, IBEA, MOEA/D, SA Abdeen et al. 2011 [10], NSGA-II Abdeen et al.2011 [11] and Bavota et al. 2013 [12] over 31 independent simulation runs. A “+” symbol at the $i^{th}$ position means that the algorithm metric median value is statistically different from the $i^{th}$ algorithm one. A “-” symbol at the $i^{th}$ position means the opposite (e.g., for Xerces-J, NSGA-III is not statistically different from IBEA, however, it is statistically different from the other algorithms).
Thus, our approach performs clearly better for remodularization solutions from developers’ stand point. To this end we reported the results of our empirical qualitative evaluation in Figure 9 (MP). As reported in Figure 9, the majority of the suggested solutions by NSGA-III improve significantly the structure (RQ1.1) while preserving the semantic coherence much better than all existing approaches. On average, for all of our five studied systems, 88% of proposed operations are considered as semantically feasible and do not generate semantic incoherence by the software engineers. This score is significantly higher than the ones of the other approaches having respectively between 51% and 70%, in average as MP scores on the different systems. Thus, our many-objective approach reduces the number of semantic incoherencies when suggesting remodularization operations. To sum up, our approach perform clearly better for semantics preservation with the cost of a slight degradation in structural improvements compared to [10][11][16]. This slight loss in the structure (RQ1.1) is largely compensated by the significant improvement of the semantic coherence.

### 5.5.2 Results for RQ1.2

To answer RQ1.2, we need to assess the correctness/meaningfulness of the suggested remodularization solutions from developers’ stand point. To this end we reported the results of our empirical qualitative evaluation in Figure 9 (MP). As reported in Figure 9, the majority of the suggested solutions by NSGA-III improve significantly the structure (RQ1.1) while preserving the semantic coherence much better than all existing approaches. On average, for all of our five studied systems, 88% of proposed operations are considered as semantically feasible and do not generate semantic incoherence by the software engineers. This score is significantly higher than the ones of the other approaches having respectively between 51% and 70%, in average as MP scores on the different systems. Thus, our many-objective approach reduces the number of semantic incoherencies when suggesting remodularization operations. To sum up, our approach perform clearly better for semantics preservation with the cost of a slight degradation in structural improvements compared to [10][11][16]. This slight loss in the structure (RQ1.1) is largely compensated by the significant improvement of the semantic coherence.

<table>
<thead>
<tr>
<th>System</th>
<th>Approach</th>
<th>NCP</th>
<th>dev.NCP</th>
<th>NP</th>
<th>dev.NP</th>
<th>NIE</th>
<th>dev.NIE</th>
<th>NAE</th>
<th>Dev.NAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xerces-J</td>
<td>NSGA-III</td>
<td>19</td>
<td>-4</td>
<td>46</td>
<td>+5</td>
<td>316</td>
<td>-69</td>
<td>432</td>
<td>+72</td>
</tr>
<tr>
<td></td>
<td>IBEA</td>
<td>21</td>
<td>-2</td>
<td>44</td>
<td>+3</td>
<td>328</td>
<td>-57</td>
<td>411</td>
<td>+51</td>
</tr>
<tr>
<td></td>
<td>MOEA/D</td>
<td>18</td>
<td>-5</td>
<td>56</td>
<td>+15</td>
<td>352</td>
<td>-33</td>
<td>397</td>
<td>+37</td>
</tr>
<tr>
<td></td>
<td>SA Abdeen et al. 2011</td>
<td>24</td>
<td>+1</td>
<td>42</td>
<td>+1</td>
<td>314</td>
<td>-71</td>
<td>441</td>
<td>+81</td>
</tr>
<tr>
<td></td>
<td>NSGA-II Abdeen et al. 2011</td>
<td>18</td>
<td>-5</td>
<td>56</td>
<td>+15</td>
<td>342</td>
<td>-43</td>
<td>437</td>
<td>+37</td>
</tr>
<tr>
<td></td>
<td>Bavota et al. 2013</td>
<td>18</td>
<td>-5</td>
<td>56</td>
<td>+15</td>
<td>302</td>
<td>-83</td>
<td>446</td>
<td>-86</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>NSGA-III</td>
<td>14</td>
<td>-6</td>
<td>38</td>
<td>+8</td>
<td>286</td>
<td>-71</td>
<td>384</td>
<td>+69</td>
</tr>
<tr>
<td>GanttProject</td>
<td>NSGA-III</td>
<td>14</td>
<td>-3</td>
<td>18</td>
<td>+9</td>
<td>259</td>
<td>-68</td>
<td>294</td>
<td>+81</td>
</tr>
<tr>
<td></td>
<td>IBEA</td>
<td>12</td>
<td>-5</td>
<td>21</td>
<td>+12</td>
<td>247</td>
<td>-80</td>
<td>304</td>
<td>+91</td>
</tr>
<tr>
<td></td>
<td>MOEA/D</td>
<td>14</td>
<td>-3</td>
<td>18</td>
<td>+9</td>
<td>259</td>
<td>-68</td>
<td>291</td>
<td>+78</td>
</tr>
<tr>
<td></td>
<td>SA Abdeen et al. 2011</td>
<td>12</td>
<td>-5</td>
<td>21</td>
<td>+12</td>
<td>238</td>
<td>-89</td>
<td>314</td>
<td>+101</td>
</tr>
<tr>
<td></td>
<td>NSGA-II Abdeen et al. 2011</td>
<td>13</td>
<td>-4</td>
<td>19</td>
<td>+10</td>
<td>244</td>
<td>-83</td>
<td>321</td>
<td>+108</td>
</tr>
<tr>
<td></td>
<td>Bavota et al. 2013</td>
<td>11</td>
<td>-9</td>
<td>47</td>
<td>+17</td>
<td>278</td>
<td>-79</td>
<td>398</td>
<td>+83</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>NSGA-III</td>
<td>16</td>
<td>-8</td>
<td>37</td>
<td>+14</td>
<td>391</td>
<td>-83</td>
<td>425</td>
<td>+84</td>
</tr>
<tr>
<td></td>
<td>IBEA</td>
<td>18</td>
<td>-6</td>
<td>33</td>
<td>+10</td>
<td>404</td>
<td>-70</td>
<td>418</td>
<td>+77</td>
</tr>
<tr>
<td></td>
<td>MOEA/D</td>
<td>16</td>
<td>-8</td>
<td>37</td>
<td>+14</td>
<td>391</td>
<td>-83</td>
<td>433</td>
<td>+92</td>
</tr>
<tr>
<td></td>
<td>SA Abdeen et al. 2011</td>
<td>14</td>
<td>-10</td>
<td>46</td>
<td>+23</td>
<td>384</td>
<td>-90</td>
<td>439</td>
<td>+98</td>
</tr>
<tr>
<td></td>
<td>NSGA-II Abdeen et al. 2011</td>
<td>15</td>
<td>-9</td>
<td>42</td>
<td>+19</td>
<td>388</td>
<td>-86</td>
<td>424</td>
<td>+83</td>
</tr>
<tr>
<td></td>
<td>Bavota et al. 2013</td>
<td>11</td>
<td>-13</td>
<td>52</td>
<td>+29</td>
<td>378</td>
<td>-96</td>
<td>438</td>
<td>+97</td>
</tr>
<tr>
<td>JDI-Ford</td>
<td>NSGA-III</td>
<td>14</td>
<td>-4</td>
<td>46</td>
<td>+21</td>
<td>301</td>
<td>-67</td>
<td>412</td>
<td>+76</td>
</tr>
<tr>
<td></td>
<td>IBEA</td>
<td>14</td>
<td>-4</td>
<td>46</td>
<td>+21</td>
<td>324</td>
<td>-44</td>
<td>422</td>
<td>+86</td>
</tr>
<tr>
<td></td>
<td>MOEA/D</td>
<td>16</td>
<td>-2</td>
<td>39</td>
<td>+16</td>
<td>308</td>
<td>-60</td>
<td>391</td>
<td>+55</td>
</tr>
<tr>
<td></td>
<td>SA Abdeen et al. 2011</td>
<td>13</td>
<td>-5</td>
<td>52</td>
<td>+27</td>
<td>297</td>
<td>-71</td>
<td>421</td>
<td>+85</td>
</tr>
<tr>
<td></td>
<td>NSGA-II Abdeen et al. 2011</td>
<td>14</td>
<td>-4</td>
<td>48</td>
<td>+23</td>
<td>304</td>
<td>-64</td>
<td>404</td>
<td>+68</td>
</tr>
<tr>
<td></td>
<td>Bavota et al. 2013</td>
<td>13</td>
<td>-5</td>
<td>52</td>
<td>+27</td>
<td>294</td>
<td>-74</td>
<td>424</td>
<td>+88</td>
</tr>
</tbody>
</table>
In addition to the empirical evaluation, we automatically evaluate our approach without using the feedback of potential users to give more quantitative evaluation to answer RQ1.2. Thus, we compare the proposed operations with some expected ones. The expected operations are those applied manually by the software engineers used in our experiments to modify an initial version of JHotDraw. We use Ref-Finder [81] to identify operations that are applied between the program version under analysis and the next version. Figure 10 summarizes our finding. We found that a considerable number of proposed operations (an average of 75% in terms of precision and recall) are already applied to the next version by software development team comparing to other existing approaches [10][11][16] having only less than 65% as precision and recall. Moreover, this score proves that our approach is useful in practice unlike both other approaches.

In conclusion, our approach produces good refactoring suggestions in terms of defect-correction ratio, semantic coherence, and code changes reduction from the point of view of 1) potential users of our refactoring tool and 2) expected refactorings applied to the next program version.

Figure 10. Quantitative evaluation (precision and recall) of the remodularization solutions (semantics)

5.5.3 Results for 1.3

To answer RQ1.3, we evaluate the number of operations (NO) suggested by the best remodularization solutions on the different systems. Figure 11 presents the code changes scores (NO) needed to apply the suggested remodularization solutions for each many-objective or multi-objective algorithm. We found that our approach succeeded in suggesting solutions that do not require high code changes (an average of only 80 operations) comparing to other many-objective (IBEA, MOEA/D) and multi-objective (NSGA-II) algorithms having respectively an average of 95 and 101 for all studied systems. We did not compare the number of changes suggested by our proposal with existing work since they are limited to only two type of changes (move class and split packages).
5.5.4 Results for RQ1.4

To answer RQ1.4, we evaluated the results of our approach comparing to other approaches that do not use the history of changes. As described in the previous sections, our NSGA-III approach outperforms clearly existing work including Abdeen et al. 2011, Abdeen et al. 2013 and Bavota et al. 2013 that are not based on the use of history of changes. This is a good indication that the recorded refactorings contribute significantly to provide good refactoring solutions. In fact, the use of history of changes is a helper objective to improve the semantic coherence of suggested remodularization solutions.

We conducted also a more quantitative evaluation to investigate effect of the use of recorded operations, on the semantic coherence (MP). To this end, we compare the MP score with and without using recorded refactorings. We present in Figure 12 the results of different combinations of our seven objectives. As presented in Figure 12, the best MP scores are obtained when the recorded code changes are considered. Moreover, we found that the optimal remodularization solutions found by our approach are obtained with a considerable percentage of reused refactoring history (ROP) (more than 75% as shown in Figure 13). Thus, the obtained results support the claim that recorded operations applied in the past are useful to generate coherent and meaningful remodularization solutions.
In the previous sections, we compared our NSGA-III proposal with one mono-objective technique [10] and one existing multi-objective technique based on NSGA-II [11]. Thus, we focus on the comparison between our NSGA-III adaption and two other many-objective algorithms IBEA and MOEA/D using the same adaptation. Table 9 shows the median IGD 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 (α < 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 7-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 3 objectives, NSGA-II performance is dramatically degraded, which is simply denoted by the ~ symbol. 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.

Figure 13. Percentage of recorded operations that are used by the best remodularization solutions

5.5.5 Results for RQ2

Table 9. Median IGD 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 (α < 1%).

<table>
<thead>
<tr>
<th>System</th>
<th>M</th>
<th>MaxGen</th>
<th>NSGA-III</th>
<th>IBEA</th>
<th>MOEA/D</th>
<th>NSGA-II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeres-J</td>
<td>3</td>
<td>250</td>
<td>9.861 x 10^4</td>
<td>9.864 x 10^4</td>
<td>9.863 x 10^4</td>
<td>9.862 x 10^4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>500</td>
<td>7.799 x 10^3</td>
<td>7.875 x 10^3</td>
<td>7.878 x 10^3</td>
<td>8.991 x 10^3</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>3</td>
<td>250</td>
<td>2.477 x 10^3</td>
<td>2.478 x 10^3</td>
<td>2.478 x 10^3</td>
<td>2.477 x 10^3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>500</td>
<td>4.193 x 10^3</td>
<td>4.201 x 10^3</td>
<td>4.206 x 10^3</td>
<td>4.533 x 10^3</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>3</td>
<td>250</td>
<td>5.536 x 10^4</td>
<td>5.801 x 10^4</td>
<td>5.796 x 10^4</td>
<td>~</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>500</td>
<td>6.099 x 10^4</td>
<td>6.208 x 10^4</td>
<td>6.193 x 10^4</td>
<td>~</td>
</tr>
<tr>
<td>GanttProject</td>
<td>3</td>
<td>250</td>
<td>5.112 x 10^3</td>
<td>5.115 x 10^3</td>
<td>5.116 x 10^3</td>
<td>5.112 x 10^3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>500</td>
<td>6.701 x 10^3</td>
<td>6.802 x 10^3</td>
<td>6.801 x 10^3</td>
<td>6.997 x 10^3</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>750</td>
<td>7.823 x 10^3</td>
<td>8.068 x 10^3</td>
<td>8.044 x 10^3</td>
<td>~</td>
</tr>
<tr>
<td>JDI-Ford</td>
<td>3</td>
<td>250</td>
<td>6.229 x 10^4</td>
<td>6.232 x 10^4</td>
<td>6.231 x 10^4</td>
<td>6.231 x 10^4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>500</td>
<td>6.608 x 10^4</td>
<td>6.682 x 10^4</td>
<td>6.686 x 10^4</td>
<td>6.887 x 10^4</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>750</td>
<td>6.984 x 10^4</td>
<td>7.305 x 10^4</td>
<td>7.299 x 10^4</td>
<td>~</td>
</tr>
</tbody>
</table>

Figure 14 illustrates the value path plots of all algorithms the 7-objective remodulairzation problem on JDI-Ford, one of 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 7 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.

![Figure 14](image)

**Figure 14.** Value path plots of non-dominated solutions obtained by NSGA-III, MOEA/D, IBEA and NSGA-II during the median run of the 7-objective remodularization problem on JDI-Ford.
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 15 shows the evolution of the running times of the different algorithms on the JDI-Ford system, one of the largest systems in our experiments. It is clear from this figure, that all the algorithms have similar running times for the 3-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 15. Computational time of the different used many-objective remodularization algorithms.](image)

5.5.6 Results for RQ3

Since it is not enough to outperform only search-based algorithms for software remodularization, we compared the results of our proposal with an existing work [16] based on the use of coupling and cohesion where only one operations type (split packages). In addition, the user needs to give as an input the different packages to restructure. We first note that the [16] (like mono-objective approaches also) provides only one remodularization solution, while NSGA-III generate a set of non-dominated solutions. In order to make meaningful comparisons, we select the best solution for NSGA-III using a knee point strategy [53]. The knee point corresponds to the solution with the maximal trade-off between all the seven objectives. Hence moving from the knee point in either direction is usually not interesting for the user [50]. Thus, for NSGA-III, 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 Bavota et al. study [16]. For the latter, we use the best solution corresponding to the median observation on 31 runs. We use the trade-off “worth” metric proposed by Rachmawati and Srinivasan [51] to find the knee point. This metric estimates the worthiness of each non-dominated refactoring solution in terms of trade-off. After that, the knee point corresponds to the solution having the maximal trade-off “worthiness” value. The results from 31 runs are depicted in Figures 8, 9 and 16, and Table 8. It can be seen that NSGA-III provides better results than [10][11][16] in all systems as discussed in the previous sections. As described in Figure 16, that NSGA-III outperforms [16] mainly because the use of history of changes as a helper objective for the semantic measures.
5.5.7 Results for RQ4

We asked the software engineers involved in our experiments to evaluate the usefulness of the suggested remodularization operations to apply one by one. In fact, sometimes these operations can improve structure and preserve the semantics but developers will consider them as not useful due to many reasons such as some packages are not used/updated anymore or includes some features that are not important, etc. Figure 17 shows that NSGA-III clearly outperforms existing work [16][10][11] by suggesting useful remodularization operations for developers. This is can be explained mainly by the use of the history of recorded changes when suggested remodularization solutions. In fact, the use of the history of changes can helps our technique to identify which packages are widely updated.

Another feature that the software engineers, involved in our experiments, found it interesting is the use of several types of remodularization operations. Figure 18 describes the distribution of the operations types used by the best solutions in all the system. It is clear that the three most important ones are move method, move class and extract/split packages. The software engineers found the idea very useful of moving methods between classes located in different packages or extracting a class then moving it to another class instead of moving the whole initial class to a new package. Sometimes, it is enough to move only a method from a class to another class in order to improve the cohesion of a package or decrease coupling between packages. However, existing remodularization work are limited to only two types of operations (move class and split packages).

During the survey, the software engineers confirm that the main limitation related to the use of NSGA-III for software remodularization is the high number of equivalent solutions that are suggested but they found the idea of the use of the Knee point as described previously useful to select a good solution. We will investigate in
our future work different other techniques to select the region of interest based on the preferences of developers.

Figure 18. Distribution of the types of suggested remodularization operations

6. 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 several quantitative measures used in our experiments. To mitigate this threat, we manually inspect and validate the remodularization solutions by a set of experts. Another threat concerns the data about the expected operations of the studied systems. In addition to the documented operations, we are using Ref-Finder which is known to be efficient [30]. Indeed, Ref-Finder was able to detect refactoring operations with an average recall of 95% and an average precision of 79% [30]. To ensure the precision, we manually inspect the refactorings found by Ref-Finder and select only those types considered in our experiments.

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 (α = 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. We identify other three threats to internal validity: selection, learning and fatigue, and diffusion.

For the selection threat, the subject diversity in terms of profile and experience could affect our study. First, all subjects were volunteers. We also mitigated the selection threat by giving written guidelines and examples of operations already evaluated with arguments and justification. Additionally, each group of subjects evaluated different operations from different systems using different techniques/algorithms.

Randomization also helps to prevent the learning and fatigue threats. For the fatigue threat, specifically, we did not limit the time to fill the questionnaire. Consequently, we sent the questionnaires to the subjects by email and gave them enough time to complete the tasks. Finally, only ten operations per system were randomly picked for the evaluation.

Diffusion threat is limited in our study because most of the subjects are geographically located in a university and a company, and the majority does not
know each other. For the ones who are in the same location, they were instructed not to share information about the experience before a certain date.

To ensure the heterogeneity of subjects and their differences, we took special care to diversify them in terms of professional status, university/company affiliations, gender, and years of experience. In addition, we organized subjects into balanced groups. This has been said, we plan to test our tool with Java development companies, to draw better conclusions. Moreover, the automatic evaluation is also a way to limit the threats related to subjects as it helps to ensure that our approach is efficient and useful in practice. Indeed, we compare our suggested operations with the expected ones that are already applied to the next releases and detected using Ref-Finder.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on five different systems belonging to different domains and with different sizes. However, we cannot assert that our results can be generalized to other applications, other programming languages, and to other practitioners. Future replications of this study are necessary to confirm the generalizability of our findings.

7. RELATED WORK

In the last two decades, a large number of research works have been proposed in the literature to support (semi-) automatic approaches to help software engineers in the re-modularization of software systems. Most of the existing approaches are based on clustering algorithms, or search-based techniques.

Wiggerts [1] provides the theoretical background for the application of cluster analysis in systems re-modularization. He discuss on how to establish similarity criteria between the entities to cluster and provide the summary of possible clustering algorithms to use in system re-modularization. Later, Anquetil et al. [2] use cohesion and coupling of modules within a decomposition to evaluate its quality. They tested some of the algorithms proposed by Wiggerts and compared their strengths and weaknesses when applied to system re-modularization. Maqbool et al. [3] focus on the application of hierarchical clustering in the context of software architecture recovery and modularization. They investigate the measures to use in this domain, categorizing various similarity and distance measures into families according to their characteristics. A more recent work by Shtern et al. [4] introduced a formal description template for software clustering algorithms. Based on this template, they proposed a novel method for the selection of a software clustering algorithm for specific needs, as well as a method for software clustering algorithm improvement. The underlying idea of these approaches is to 1) group in a module highly cohesive source code components, where the cohesiveness is measured in terms of intra-module links; and 2) reduce the coupling between modules, where the coupling is measured in terms of inter-module dependencies. The vast majority of clustering-based approaches aim at modularizing/decomposing software systems from scratch using only structural measures. However, the goal of our approach aims at assisting software maintainers in the task of improving the quality of existing packages structure while maintaining the semantic coherence of the original design structure.

There have been several developments in search-based approaches to support the automation of software modularization. Mancoridis et al. [5] was the first search-based approach to address the problem of software modularization using a single-objective approach. Their initial work [5] to identify the modularization of a software system is based hill-climbing to maximize cohesion and minimize coupling. The same technique has been also used in [6] and [7] where the authors present Bunch [6], a tool supporting automatic system decomposition. Subsystem decomposition is
performed by Bunch by partitioning a graph of entities and relations in a given source code. To evaluate the quality of the graph partition, a fitness function is used to find the trade-off between interconnectivity (i.e., dependencies between the modules of two distinct subsystems) and intraconnectivity (i.e., dependencies between the modules of the same subsystem), to found out a satisfactory solution. In [8], Harman et al. use a genetic algorithm to improve the subsystem decomposion of a software system. The fitness function to maximize is defined using a combination of quality metrics, e.g., coupling, cohesion, and complexity. Similarly, Seng et al. [9], treated the re-modularization task as a single-objective optimization problem using genetic algorithm. The goal is to develop a methodology for object-oriented systems that, starting from an existing subsystem decomposition, determines a decomposition with better metric values and fewer violations of design principles. Abdeen et al. [10] proposed an heuristic search-based approach for automatically optimizing (i.e., reducing) the dependencies between packages of a software system using simulated annealing. Their optimization technique, is based on moving classes between the original packages. Taking inspiration from our previous work [14][76][77], Abdeen et al. have recently extended their initial work to consider the re-modularization task as a multi-objective optimization problem to improve existing packages structure while minimizing the modification amount on the original design [11]. Using NSG-II, this optimization approach aims at increasing the cohesion and reducing the coupling and cyclic connectivity of packages, by modifying as less as possible the existing packages organization. Praditwong et al. [15] have recently formulated the software clustering problem as a multi-objective optimization problem. Their work aim at maximizing the modularization quality measurement [5]; minimizing the inter-package dependencies; increasing intra-package dependencies; maximizing the number of clusters having similar sizes; and minimizing the number of isolated clusters.

Most of the re-modularization approaches in the literature are based on information derived only from structural metrics to modularize/restructure the original package organization. However, this is not enough to produce a semantically coherent design. The first attempt that addresses this problem was by Bavota et al. [12] [16] who proposed an automated, single-objective, approach to split an existing package into smaller but more cohesive ones. The proposed approach analyzes the structural and semantic relationships between classes in a package identifying chains of strongly related classes. The identified chains are used to define new packages with higher cohesion than the original package. This work has been extended in [13], to propose an interactive multi-objective re-modularization approach. The proposed Interactive Genetic Algorithms (IGAs) aims at integrating developer's knowledge in a re-modularization task. Specifically, the proposed algorithm uses a fitness composed of automatically-evaluated factors (accounting for the modularization quality achieved by the solution) and a human-evaluated factor, penalizing cases where the way re-modularization places components into modules is considered meaningless by the developer. One of the limitations of this approach is that, in each generation of the re-modularization process, end users should analyze the suggested solution, class-by-class and package-by-package, and provide their feedback. User feedback can be either about classes which should stay together, or not, and/or about small/isolated clusters. This is not always profitable when we deal with industrial size software projects, and it need expert users to suitably drive the optimization process.

The semantic meaningfulness of the recommended restructuration (e.g., re-modularization, refactoring) is a fundamental issue when automatically modifying a software design. The first attempt to integrate the semantic coherence of when automatically modifying the software design was in [17]. Similarly, our re-
modularization approach use the combination of semantic and structural information captured in the package and class levels to suggest more meaningful re-modularization and better decide how to group together, split, or move, (or not) certain code elements. Furthermore, while automatic re-modularization approaches proved to be very effective to increase cohesiveness and reduce coupling of software modules, they do not take into account the history of changes that provide a lot of information that is very useful in automating many software maintenance tasks. One of the characteristics of our approach is that it exploits the change history that represents an effective way to produce more meaningful re-modularizations. Another issue is that the majority of existing re-modularization approaches considers only moving classes or grouping/splitting packages; however, none considered move methods/fields among classes in different packages. Hence, sometimes, it is enough to move only a method or a field between two classes in two different packages to reduce the dependency between them.

8. CONCLUSIONS AND FUTURE WORK

In this paper we introduced a new scalable search-based software engineering approach for software remodularization based on NSGA-III. This paper represents the first real-world application of NSGA-III and the first scalable work that supports the use of 7 objectives to address a software engineering problem. We address several challenges of existing software remodularization techniques that are limited to mainly the use of coupling and cohesion, and few types of operations (move class and split package). Our proposal aims at finding the remodularization solution that improve the structure of packages by optimizing some metrics such as number of classes per package, number of packages, coupling and cohesion; improve the semantic coherence of the restructured program; minimize code changes; and maximize the consistency with development change history.

We evaluated our approach on four open source systems and one industrial system provided by our industrial partner Ford Motor Company. We report the results on the efficiency and effectiveness of our approach, compared to the state of the art remodularization approaches. Our results indicate that our approach significantly outperforms, in average, existing approaches based on a quantitative and qualitative evaluation. All the results were statistically significant on the 31 independent simulations using the Wilcoxon rank sum test [41] with a 99% confidence level (α < 1%) where more than 92% of code-smells were fixed on the different open source systems.

As part of the future work, we plan to work on adapting NSGA-III to additional software engineering problems and we will perform more comparative studies on larger open source systems. Furthermore, we will investigate the impact of different parameter settings on the quality of our results. Nevertheless, this extensive study has shown a direction using NSGA-III to handle as many as 7 objectives in the context of solving software engineering problems and would remain as one of the first studies in which such a large number of objectives have been considered. Finally, we plan to extend the use of our modularization approach by additional experts to generalize the obtained results.

REFERENCES


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


[77] Ali Ouni, Marouane Kessentini, Houari A. Sahraoui: Search-Based Refactoring Using Recorded Code Changes, CSMR 2013: 221-230


