Search-based Refactoring: Towards Semantics Preservation

Ali Ouni[1,2], Marouane Kessentini[2], Houari Sahraoui[1] and Mohamed Salah Hamdi[3]
[1]DIRO, Université de Montréal, Canada
{ouniali, sahraouh}@iro.umontreal.ca
[2]CS, Missouri University of Science and Technology, USA
marouanek@mst.edu
[3]Information Systems Department, Ahmed Ben Mohamed Military College, Qatar
mshamdi@yahoo.com

***Abstract*—Refactoring restructures a program to improve its structure without altering its behavior. However, it is challenging to preserve the domain semantics of a program when refactoring is decided/implemented automatically. Indeed, a program could be syntactically correct, have the right behavior, but model incorrectly the domain semantics. In this paper, we propose a multi-objective optimization approach to find the best sequence of refactorings that maximizes quality improvements (program structure) and minimizes semantic errors. To this end, we use the non-dominated sorting genetic algorithm (NSGA-II) to find the best compromise between these two conflicting objectives. We report the results of our experiments on different open source projects.**

Keywords- Search-based Software Engineerng; Software Maintenance; Design-defects; Multi-objective Optimization; Semantic Similarity

I. INTRODUCTION

In object-oriented (OO) programs, objects reify domain concepts and/or physical objects. They implement their characteristics and behavior. Unlike for 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 during maintenance, the adequacy with domain semantics could be compromised. Indeed, over the past decades, many techniques and tools have been developed in order to improve design quality using refactoring. The majority of existing contributions have formulated the refactoring recommending problem as a single-objective optimization problem, in which the goal is to maximize code quality while preserving the behavior (see for example, [3], [4], [1], [11], and [2]). However, one of the challenges, when maximizing quality, is to preserve the domain semantics. Let us consider the example of a refactoring solution that moves a method "calculateSalary" from the class "Employee" toward the class "Car". This refactoring could improve the program structure by reducing the number of methods in "Employee" and satisfies the pre- and post-conditions to preserve the behavior. However, having a method "calculateSalary" in "Car" does not make sense from the domain semantics standpoint.

In previous work [2], we proposed an approach that recommends refactoring solutions to correct design defects and minimize code modification effort using a multi-objective optimization approach. In this paper, our aim is to improve the quality of suggested refactoring solutions that are generated using our search-based approach, by minimizing domain semantics errors. To this end, we use a multi-objective optimization algorithm to propose refactoring sequences that maximize software quality while minimizing the impact on domain semantics. The proposed algorithm is an adaptation of the Non-dominated Sorting Genetic Algorithm (NSGA-II) [9], a multi-objective evolutionary algorithm (MOEA) [8]. NSGA-II aims at finding a set of representative Pareto optimal solutions in a single run. The evaluation of these solutions is based on conflicting criteria. To preserve the domain semantics during the refactoring process, two techniques are combined to estimate the semantic proximity between classes when moving elements between them. The first technique evaluates the cosine similarity of the used vocabulary (names of methods, fields, types, super and sub classes, etc.). The second technique considers the dependencies between classes extracted from call graphs.

The remainder of this paper is structured as follows. Section 2 is dedicated to the background needed to understand our approach and the refactoring challenges. In Section 3, we describe how the semantic similarity is calculated among software elements to guide the search process. Then, in Section 4, we give an overview of our proposal, and we explain how we adapted the non-dominated sorting genetic algorithm to find "good" refactoring strategies. Section 5 presents and discusses the evaluation results. The related work in search-based refactoring and conceptual similarity is outlined in Section 6. We conclude and suggest future research directions in Section 7.

978-1-4673-2312-3/12/$31.00 ©2012 IEEE

II. Background and Refactoring Challenges

In this section, we present first some definitions. Then we describe the different challenges addressed in this paper.

Opdyke [12] defines refactoring as the process of improving code after it has been written by changing its internal structure without changing the external behavior. One motivation is to reorganize fields, classes and methods in order to facilitate future extensions. This reorganization is used to improve different aspects of software-quality: reusability, maintainability, complexity, etc. [12]. As defined in [20], the refactoring process includes six main steps: (1) identify refactoring opportunities, (2) determine which refactorings to apply, (3) ensure that applied refactorings preserve the behavior, (4) execute selected refactorings, (5) evaluate the impact of applying the refactorings on the system quality, and (6) guarantee the coherence of non-code artefacts such in documentation, requirements, and tests.

In this paper, we do not focus on the first step related to the detection of refactoring opportunities. We consider that different design defects are already detected, and need to be corrected. These design defects, also called anomalies, flaws, bad smells, or anti-patterns, refer to design situations that adversely affect the development of software. As stated by Fenton and Pfleeger [7], design defects are unlikely to cause failures directly, but may do it indirectly. In general, they make a system difficult to change, which may in turn introduce bugs. The most well-known example of defects is the Blob. It is found in designs where one large class monopolizes the behavior of a system (or part of it), and other classes primarily encapsulate data. The detected defects can be fixed by applying some refactoring operations. For example, to correct the blob defect many operations can be used to reduce the number of functionalities in a specific class, such as moving methods and/or extracting a class.

In this paper, widely-used refactoring operations are considered. Each of these refactoring operations takes a list of controlling parameters as illustrated in Table 1. For example, as described in Figure 1, the operation *MoveMethod(TaskImpl, TaskManagerImpl, getSuperTask())* indicates that the method "*getSuperTask()*" is moved from the source class "*TaskImpl*" to the target class "*TaskManagerImpl*".

Table 1. List of refactoring operations and its controlling parameters

Refactorings	Controlling parameters
move method	(sourceClass, targetClass, method)
move field	(sourceClass, targetClass, field)
pull up field	(sourceClass, targetClass, field)
pull up method	(sourceClass, targetClass, method)
push down field	(sourceClass, targetClass, field)
push down method	(sourceClass, targetClass, method)
inline class	(sourceClass, targetClass)
extract class	(sourceClass, newClass)

Even though most of the existing refactoring approaches are powerful enough to provide solutions for the different six steps, some issues are still to be addressed. Indeed, it is important to ensure that, after applying refactorings, the program design is consistent with the domain semantics. The definition of semantic consistency, used here, is based on the original meaning to stick together. We want to stress the fact that there is a reason why and how code elements are grouped and connected. Thus, the semantic coherence should be preserved, and not only the behavior, when applying refactorings to restructure programs.

The use of structural information is not sufficient to ensure the semantic coherence. Some refactorings could improve code structure but model domain semantics incorrectly. Figure 1 illustrates an example, extracted from the open-source system Gantt [24], where a *move method* operation is applied to move the method *getSuperTask()* from class *TaskImpl* to *GanttCVSExport*. This code change is semantically incorrect since *GanttCVSExport* functionality is related to the conversion of CVS files and not to task management. However, we improve the design quality because we reduce the number of method in the class *TaskImpl* detected as a blob. A good refactoring operation should move the method *getSuperTask()* from class *TaskImpl* to *TaskManagerImpl* since the two classes are semantically close.

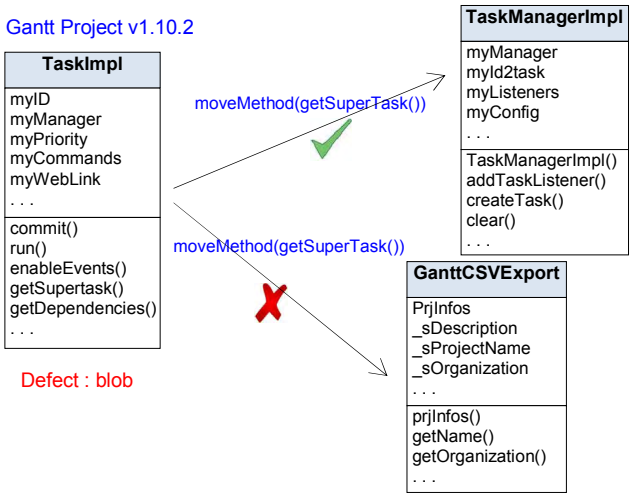

Fig 1. Motivating example: semantic incoherence

In most of the existing work related to automated refactorings, the semantic coherence is not a major concern. The refactoring process needs a manual inspection by the user to evaluate the meaningfulness/feasibility of proposed refactorings. The inspection aims to verify if these refactorings may not produce semantic incoherence in the program design. For large-scale systems, this manual inspection is complex, time-consuming and error-prone.

Another observation is that improving quality (code structure) and minimizing semantic incoherence are complementary and sometimes conflicting considerations.

In some cases, improving the program structure could provide a design that does not make sense semantically. For this reason, we need to find a compromise between improving code structure and reducing the semantic incoherence.

These observations are at the origin of the work described in this paper that will be described in the next section.

III. APPROXIMATING SEMANTIC SIMILARITY

The solution proposed in this paper is based on the semantic similarity between code elements when applying refactoring operations. Semantic similarity is captured by different measures that could be integrated into existing refactoring approaches to help preserving domain semantics. These measures are classified into two categories: 1) vocabulary-based similarity; and 2) dependency-based similarity. In this section, we describe how this semantic similarity is calculated.

A. Vocabulary-based similarity

We start from the assumption that the vocabulary of software elements is borrowed from the domain terminology and then could be used to determine which part of the domain semantics is encoded by an element. Thus, two software elements could be semantically similar if they use a similar/common vocabulary. The vocabulary of an element includes names of methods, fields, variables, parameters, type declaration, etc. This similarity could be interesting to consider when moving methods, renaming fields/methods/classes, etc. For example, when a method has to be moved from one class to another, the refactoring would probably make sense if both classes share the same vocabulary.

Approximating the domain semantics with vocabulary is widely used in several areas, *e.g.*, information retrieval, natural language processing, and other related areas. In all these techniques, the semantic similarity between two entities indicates if they share many common elements (properties, words, etc.). For our proposal, we use the cosine similarity measure which has been well studied to estimate the semantic similarity between documents [22]. Documents are represented as vectors of terms in *n*-dimensional space where *n* is the number of different terms in the document. For each document, a weight is assigned to each dimension (representing specific term) of the representative vector that corresponds to the term frequencies score (TF) in the document. The similarity between documents is measured by the cosine of the angle between its representative vectors as a normalized projection of one vector over the other. The cosine measure between a pair of document A and B is defined as follows:

$$Sim(A,B) = \cos(\theta) = \frac{\vec{A} * \vec{B}}{\left|\vec{A}\right| * \left|\vec{B}\right|} \quad (1)$$

By analogy, a software element could be considered as a document. Therefore, using program analysis frameworks such as Soot [21], the source code can be analyzed in order to extract the used vocabulary for each element in the program (*e.g.*, class, method, etc.). In general, the vocabulary related to a software element is represented by the identifiers/names of class, methods, fields, parameters, type declaration, etc. Then, each element will be represented as a vector in *n*-dimensional space where *n* is the number of code elements.

After the vocabulary extraction step, the TF (term frequency) score is calculated. TF represents the coefficients of the vector for a given element. Hence, TF could be calculated by suitable static analysis of elements in the code. Then, the distance between two vectors is considered as an indicator of its similarity. Therefore, using cosine similarity, the conceptual similarity among a pair of elements *c1* and *c2* is quantified by

$$Sim(c1,c2) = \frac{\sum_{i=0}^{n-1} \left(tf_{c1}(w_i) * tf_{c2}(w_i)\right)}{\sqrt{\sum_{i=0}^{n-1} \left(tf_{c1}(w_i)\right)^2} \sqrt{\sum_{i=0}^{n-1} \left(tf_{c2}(w_i)\right)^2}} \quad (2)$$

where $tf_{c1}(w_i)$ and $tf_{c2}(w_i)$ are term frequency values of the term w_i in the representative vectors of elements *c1* and *c2*.

To improve the semantic similarity measurement among software elements, we use a second heuristic to investigate the degree of dependencies between the elements.

B. Dependency-based similarity

In addition to the vocabulary similarity, another heuristic is used to approximate domain semantics closeness between two elements starting from their mutual dependencies. The intuition is that elements that are strongly connected (i.e., having dependency links) are semantically related. As a consequence, refactoring operation requiring semantics closeness between involved elements are likely to be successful when these elements are strongly connected.

In our measures, we considered two types of dependency links: 1) shared method calls, and 2) shared field access. These dependency links could be captured using classical static and/or dynamic analysis techniques.

1) Shared method call

In a first step, a method call graph is derived. A call graph is a directed graph which represents the different calls among all methods of the entire program, in which nodes represent methods, and edges represent calls between these methods. Then, for a pair of elements, shared calls are derived from this graph. We distinguish between shared call-out and shared call-in. Equations (3) and (4) are used to measure respectively the shared call-out and the shared call--in between two elements c1 and c2 (classes, for example).

$$\text{sharedCallOut}(c1,c2) = \frac{|\text{callOut}(c1) \cap \text{callOut}(c2)|}{|\text{callOut}(c1) \cup \text{callOut}(c2)|} \quad (3)$$

$$\text{sharedCallIn}(c1, c2) = \frac{|\text{callIn}(c1) \cap \text{callIn}(c2)|}{|\text{callIn}(c1) \cup \text{callIn}(c2)|} \quad (4)$$

A shared method call is defined as the weighted sum of shared call-in and call-out, as illustrated in Equation (5)

$$\text{sharedMethodCall}(c1, c2) = \alpha * \text{sharedCallOut}(c1, c2) + \beta * \text{sharedCallIn}(c1, c2) \quad (5)$$

with $\alpha + \beta = 1$

2) Shared field access

Our approach uses static analysis to identify element dependencies based on field accesses (read or modify). We start from the hypothesis 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 elements c1 and c2 is calculated according to Equation (6). In this equation, accessFieldRW(c_i) computes the number of fields that may be read or modified by each method of the element c_i. Thus, by applying a suitable static analysis to the whole method body, all field references that occur could be captured.

$$\text{sharedFieldsRW}(c1, c2) = \frac{|\text{accessFieldsRW}(c1) \cap \text{accessFieldsRW}(c2)|}{|\text{accessFieldsRW}(c1) \cup \text{accessFieldsRW}(c2)|} \quad (6)$$

The next section will describe how the different semantic measures described in this section are used in our heuristic-search adaptation, mainly in the fitness function definition.

IV. SEARCH-BASED REFACTORING USING MULTI-OBJECTIVE OPTIMIZATION

In this section, we describe our proposal, and how we consider the refactoring as a multi-objective optimization problem. We start by giving an overview of our approach. Then, we detail our algorithm adaptation, including: solution representation, change operators, and fitness functions.

A. Approach overview

In this paper, an automated approach is proposed, to ameliorate the quality of a system while preserving its domain semantics. This approach uses a multi-objective optimization to find the best compromise between code quality improvements and domain semantics preservation. This results into two conflicting objectives:

1) Improving software quality corresponds to correcting design defects. We use defect detection rules, proposed in our previous work [2] , to find the best refactoring solutions, from an exhaustive list of refactoring operations, which minimize the number of detected defects.

2) Preserving the semantic coherence after applying the suggested refactorings is ensured by maximizing the different semantic measures score described in the previous section.

The general structure of our approach is sketched in Figure 2. The search-based process takes as inputs, the source code with defects, detection rules, a set of refactoring operations, and a call graph for the whole program. As output, our approach recommends refactoring solutions. A solution consists of a sequence of refactoring operations that should be applied to correct the input code. The used detection rules are expressed in terms of metrics and threshold values. Each rule represents a specific defect (e.g., blob, spaghetti code, functional decomposition) and is expressed as a combination of a set of quality metrics/threshold values [2].

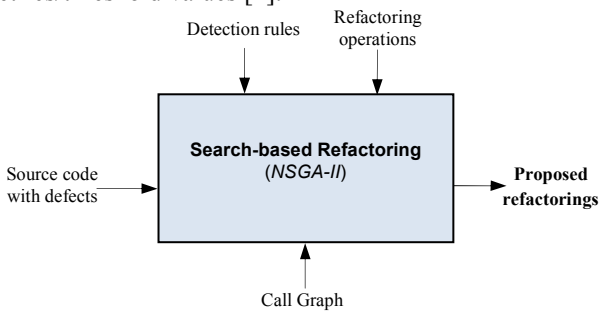

Fig 2. Approach overview

Then the process of generating a correction solution can be viewed as the mechanism that finds the best list among all available refactoring operations that best reduces the number of detected defects and at the same time preserves the domain semantics of the initial program. In other words, this process aims at finding the best tradeoff between the two conflicting criteria.

As far as the search-based refactoring is concerned, the size of the search space is determined not only by the number of possible refactoring combinations, but also by the order in which they are applied. Due to the large number of possible refactoring solutions [10] and the conflicting objectives related to the quality improvements and the semantics preservation, we considered the refactoring as a multi-objective optimization problem instead of a single-objective one. To this end, we adapted the non-dominated sorting genetic algorithm (NSGA-II) [9].

B. NSGA-II for search-based refactoring

1) NSGA-II overview

NSGA-II is a powerful search method that simulates the natural selection that is inspired from the theory of Darwin [9]. Hence, the basic idea is to make a population of candidate solutions evolve toward the best solution in order to solve a multi-objective optimization problem. NSGA-II was designed to be applied to an exhaustive list of candidate solutions, which creates a large search space.

The main idea of the NSGA-II is to calculate the Pareto front that corresponds to a set of optimal solutions, called non-dominated solutions, or also Pareto set. A non-dominated solution is the one which provides a suitable compromise between all objectives without degrading any of them. Indeed, the concept of Pareto dominance consists of comparing each solution *x* with every other solution in the population until it is dominated by one of them. If any solution does not dominate it, the solution *x* will be

considered non-dominated and will be selected by the NSGA-II to be one of the set of Pareto front. If we consider a set of objectives f_i , $i \in 1\ldots n$, to maximize, a solution x dominates x'

$$iff\ \forall i, f_i(x') \leq f_i(x) \text{ and } \exists j \mid f_j(x') < f_j(x).$$

The first step in NSGA-II is to create randomly the initial population P_0 of individuals encoded using a specific representation. Then, a child population Q_0 is generated from the population of parents P_0 using genetic operators such as crossover and mutation. Both populations are merged and a subset of individuals is selected, based on the dominance principle to create the next generation. This process will be repeated until reaching the last iteration according to stop criteria.

2) NSGA-II Adaptation

We describe in this section how we adapted the NSGA-II to find the best tradeoff between quality and semantics dimensions. As our aim is to maximize the quality and minimize semantic errors, we consider each one of these criteria as a separate objective for NSGA-II. The pseudo-code for the algorithm is given in Algorithm 1. The algorithm takes as input the whole source code to be corrected, a set of possible refactoring operations RO, defect detection rules D, and a call graph. Lines 1–5 construct an initial population based on a specific representation, using the list of RO (described in table 1) given at the input. Thus, the initial population stands for a set of possible defect-correction solutions represented as sequences of RO which are randomly selected and combined.

Lines 6–22 encode the main NSGA-II loop whose goal is to make a population of candidate solutions evolve toward the best sequence of RO, *i.e.,* the one that minimizes as much as possible the number of defects and preserves the domain semantics. During each iteration t, a child population Q_t is generated from a parent generation P_t (line 7) using genetic operators. Then, Q_t and P_t are assembled in order to create a global population R_t (line 8). Then, each solution S_i in the population R_t is evaluated using our two fitness functions, quality and semantic (lines 11 and 12):

- *Quality fitness function* (line 11) to maximize: represents the number of detected defects after applying the proposed refactoring sequence.
- *Semantic incoherence fitness function* (line 12) to minimize: calculates the semantics similarity of the elements to be changed by the refactoring using the semantic measures described in section 3.

Once Quality and Semantics are calculated, all the solutions will be sorted in order to return a list of non-dominated fronts F (line 14). When the whole current population is sorted, the next population P_{t+1} will be created using solutions that are selected from sorted fronts F (lines 16-19). When two solutions are in the same front, *i.e.*, same dominance, they are sorted by the crowding distance, a measure of density in the neighborhood of a solution [9]. The algorithm terminates (line 22) when it achieves the termination criterion (maximum iteration number). The algorithm returns the best solutions that are extracted from the first front of the last iteration (line 23).

We give more details in the following sub-sections about the representation of solutions, genetic operators, and the fitness functions.

Algorithm: Search-based Refactoring

Input:
defect _code : source code with design defects,
Set of refactoring operations RO,
Defect detection rules D,
cg : Call graph

Process:
1. initial_population(P, Max_size)
2. P_0:= set_of(S)
3. S:= sequence_of(RO)
4. code:= defect_Code
5. t:=0
6. repeat
7. Q_t:= Gen_Operators(P_t)
8. R_t:=P_t U Q_t
9. for all $S_i \in R_t$ do
10. code:= execute_refactoring(S, defect_Code);
11. *Quality*(S_i) := calculate_Quality(D, code);
12. *Semantic*(S_i) := calculate_Sematics(code, cg);
13. end for
14. F:=fast-non-dominated-sort(R_t)
15. P_{t+1} :=Ø
16. while $|P_{t+1}|$<Max_size
17. F_i := crowding_distance_assignment(F_i)
18. P_{t+1} := P_{t+1}+F_i
19. end while
20. P_{t+1} :=P_{t+1}[0:Max_size]
21. t:=t+1;
22. until t=max_it
23. best_solution = First_front(R_t)
24. return best_solution

Output:
best_solution: Near-Optimal refactoring sequences

Algorithm 1. High-level pseudo-code for NSGA-II adaptation to our problem

a) Solution Representation

To represent a candidate solution (individual), we used a vector representation. Each vector's dimension represents a refactoring operation. When created, the solution will be executed in the vector order to be evaluated. An example of a solution is given in Figure 3. Of course, for each of these refactorings we specify pre- and post-conditions that are already studied in [12] to ensure the feasibility of applying them. These constraints are also used when change operators are applied. In addition, the order of applying these refactorings corresponds to their positions in the vector.

RO1	moveMethod
RO2	pullUpAttribute

RO3	extractClass
RO4	inlineClass
RO5	extractSuperClass
RO6	inlineMethod
RO7	extractClass
RO8	moveMethod

Fig 3. Representation of an NSGA-II individual

b) Selection and genetic operators

Selection. There are many selection strategies where fittest individuals are allocated more copies in the next generations than the other ones. Thus, to guide the selection process, NSGA-II uses a comparison operator based on a calculation of the crowding distance to select potential individuals to construct a new population P_{t+1}. Furthermore, for our initial prototype, we used Stochastic Universal Sampling (SUS) to derive a child population Q_t from a parent population P_t, in which each individual's probability of selection is directly proportional to its relative overall fitness value (average score of the two fitness values) in the population. We use SUS to select elements from P_t that represents the best elements to be reproduced in the child population Q_t using genetic operators such as mutation and crossover.

Mutation. Mutation of a solution starts by randomly selecting one or more operations from its associated sequence. Then, the selected operation(s) are replaced by other ones from the initial list of in the RO set. An example is shown in figure 4.

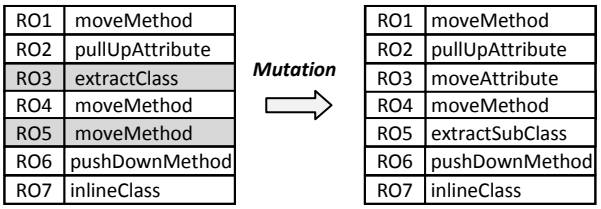

Fig 4. Mutation operator

Crossover. We use a single, random, cut-point crossover. The two selected parent solutions are split in two parts. Then crossover creates two child solutions by putting, for the first child, the first part of the first parent with the second part the second parent, and, for the second child, the first part of the second parent with the second part of the first parent. This operator must ensure the respect of the length limits by eliminating randomly some refactoring operations. As illustrate in figure 5, each child combines some of the refactoring operations of a parent with some ones of the second parent. In any given generation, each solution will be the parent in at most one crossover operation.

c) Multi-criteria evaluation (Fitness Functions)

In the majority of existing work, the fitness function maximizes the quality of the source code. In this work, we distinguish between two different fitness functions to include in our NSGA-II adaptation: (1) the quality of the corrected code, and (2) the semantic coherence of changes.

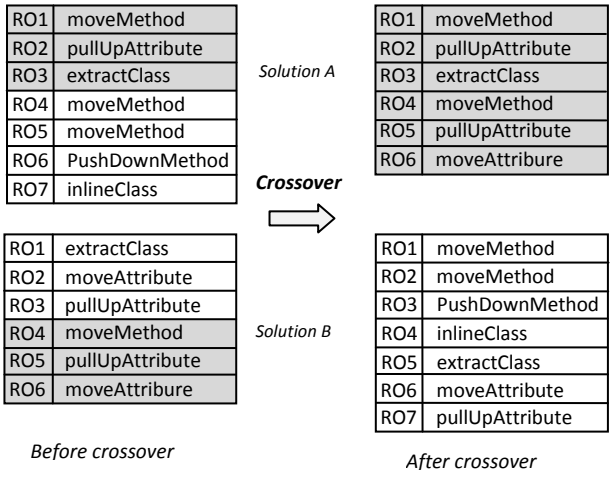

Fig 5. Crossover operator

Quality criterion

The Quality criterion is evaluated using the fitness function given in Equation (7). The quality value increases when the number of defects in the code is reduced after the correction. This function returns a real value between 0 and 1 that represents the proportion of defected classes (detected using design defects detection rules D) compared to the total number of possible defects that can be detected. The detection of defects is based on some rules defined in our previous work [2].

$$Quality = \frac{\#corrected\_defects}{\#detected\_defects\_before\_refactoring} \quad (7)$$

Semantic criterion

To formulate the semantic fitness function, we use and combine the semantic measures described in section 3 to calculate the semantic coherence of each change proposed by a refactoring operation.

Thus, the semantics $Sem(RO_i)$ of a refactoring operation RO_i corresponds to the weighted sum of each measure (vocabulary and dependency-based similarity) used to calculate the similarity between modified code elements (after applying a refactoring operation). Hence, the semantic fitness function of a solution corresponds to the average of semantic coherence for each applied refactoring

$$Semantic = \frac{\sum_{i=0}^{n-1} Sem(RO_i)}{n} \quad (8)$$

where n is the size of the solution, i.e., the number of refactoring operations.

To illustrate the fitness function, we suppose that a solution contains only one refactoring: Extract Class. We

choose this operation because it is the most difficult refactoring to guarantee its semantics coherence after applying it. The goal of this refactoring is to find a cohesive set of methods and fields to be extracted from a source class and to move this set to a newly created class. The best set maximizes the cohesion of the newly extracted class and minimizes the coupling with the source class. To this end, we study the intra-element similarity (i.e., classes) in order to find a set of methods and fields that satisfy these criteria. In fact, when applying the extract class refactoring on a specific class it will be splitted in two classes. Thus, we need to calculate the semantic similarity between elements in the original class to decide how to split it in two classes.

We use our two measures to calculate the semantic similarity between the class elements (methods and fields). Let's consider a source class that contains n methods $\{m_1,\ldots m_n\}$ and m fields $\{f_1,\ldots f_m\}$. The idea is to calculate the similarity between each element method-field and method-method in a matrix as shown in table 2.

Table 2. An example intra-element similarity matrix

	f_1	f_2	…	f_m	m_1	m_2	…	m_n	average
m_1	1	0		1	1	0.15		0.1	0.42
m_2	**0**	**1**		**1**	**1**	**1**		**0**	**0.6**
. . .									
m_n	1	0		0	0.6	0.2		1	0.32

The intra-element similarity matrix would be calculated as following: for the similarity between method-method, we consider both similarity measurements. However for the similarity method-field we use only the second measure related to shared field, i.e., if the method m_i may access (read or write) to the field f_j then the similarity value is 1 and 0 otherwise. The column average contains the average of similarity values for each line. Thus, the suitable set of methods and fields to be moved to the new class corresponds to the line which has a higher average value. Hence, the elements that should be moved are those which have a similarity value higher than a threshold value of 0.5.

V. EVALUATION

To evaluate our proposal, we conducted experiments on two open-source systems. In this section, we present the objectives of this exploratory study, and we describe and discuss the obtained results.

A. Research Questions

The goal of our study is to find out whether our approach could find meaningful sequences of refactorings to correct defects while preserving the domain semantics, from the perspective of a software maintainer conducting a quality audit. Indeed, our study addresses two research questions, which are defined here. We also explain how our experiments are designed to address them. The two research questions are then:

- **RQ1:** To what extent can the proposed approach correct design defects and preserve the domain semantics when applying refactorings?
- **RQ2:** To what extent can the semantics preservation improve the results provided by existing work?

To answer RQ1, we validate manually the proposed refactoring operations to fix defects. To this end, we calculate the number of defects that are corrected after applying the best solutions. Then, we verify the semantic coherence of the code changes proposed by each refactoring.

To answer RQ2, we compared our results to those produced by two existing contributions [1], [3]. In [3], Harman et al. proposed a multi-objective approach that uses two quality metrics to improve (coupling between objects and standard deviation of methods per class), after applying the refactorings sequence. In [1], we used a single-objective genetic algorithm to correct defects by finding the best refactoring sequence that reduce the number of defects. In both contributions, the semantics preservation is not addressed explicitly. We evaluate if our refactoring solutions produces more coherent changes than those proposed by [1] and [3] while having similar quality improvement.

B. Setting

We used two open-source Java projects to perform our experiments: GanttProject (Gantt for short) v1.10.2 [24] and Xerces-J v2.7.0 [25]. Table 3 provides some descriptive statistics about these two programs.

Table 3. Program statistics.

Systems	Number of classes	KLOC	Number of Defects
GanttProject v1.10.2	245	31	41
Xerces-J v2.7.0	991	240	66

We chose Xerces-J and Gantt because they are medium-sized open-source projects and were analysed in the related work. The version of Gantt studied was known to be of poor quality, which has led to a new major revised version. Xerces-J is in active development since the past 10 years, and their design has not been responsible for a slowdown of their developments.

To answer our two research questions, we derived two metrics: defect correction ratio (DCR) and refactoring precision (RP). DCR is given by Equation (9) and calculates the number of corrected defects over the total number of defects detected before applying the proposed refactoring sequence.

$$DCR = \frac{\#\text{ corrected defects}}{\#\text{ defects before applying refactorings}} \quad (9)$$

For the refactoring precision (RP), we manually inspected the semantic correctness of the proposed refactoring operations for each system. We applied the proposed refactoring operations using ECLIPSE [23] and we checked the semantic coherence of modified code fragments. RP is then equal to the number of meaningful refactoring operations, in terms of semantic coherence, over the total number of suggested ones. RP is given by (10)

$$RP = \frac{\#\text{ meaningful refactorings}}{\#\text{proposed refactorings}} \quad (10)$$

C. Results and Discussions

As described in Table 4, the majority of proposed refactoring sequences improve code quality with acceptable scores compared to existing work. In addition, we preserve the semantics better than the two other approaches. For Gantt, 128 of the 146 proposed refactoring operations (91%) do not generate semantic incoherence. This score is higher than one of the two other approaches having respectively 69% and 73% as RP scores. Thus, our multi-objective approach reduces the number of semantic incoherence when applying refactoring operations. In the same time, after applying the proposed refactoring operations, we found that more than 87% (36/41) of detected defects were fixed. The corrected defects were of different types (blob, spaghetti code, and functional decomposition [1]). The majority of non-fixed defects are related to the blob type. This type of defect usually requires a large number of refactoring operations and is then very difficult to correct. This score is comparable to the correction score of Kessentini et al.'s approach (92%). Thus, the slight loss in quality is largely compensated by the improvement of the semantic coherence.

We had similar results for Xerces-J. The majority, 78%, of defects was corrected (51/66) and most of the proposed refactoring sequences, 89% (197/221), are coherent semantically. When comparing with previous work, Kessentini et al. performs slightly better for the number of corrected defects than NSGA-II (89%). For the refactoring precision, strategies proposed by NSGA-II require less manual adaptation than those of Kessentini et al. Indeed, the majority of refactoring operations proposed by NSGA-II does not need programmer intervention to solve semantic incoherencies that are introduced when applying them. 89% of proposed refactorings by NSGA-II for Xerces-J are correct, whereas Kessentini et al. and Harman et al. proposes respectively 69% and 63% of good refactoring operations.

In conclusion, our approach produces good refactoring suggestions both from the point of views of defect-correction ratio and semantic coherence. These results are comparable to those obtained by approaches that consider only the quality without considering the semantic coherence.

Table 4. Refactoring results.

Systems	Approach	DCR	RP
GanttProject	NSGA-II	87% (36|41)	91% (128|146)
v1.10.2	Harman et al. 07	N.A	69% (218|312)
	Kessentini et al. 11	95% (39|41)	73% (158|216)
Xerces-J	NSGA-II	78% (51/66)	89% (197|221)
v2.7.0	Harman et al. 07	N.A	63% (262|417)
	Kessentini et al. 11	89% (59/66)	69% (212|304)

Another element that should be considered when comparing the results of the three algorithms, is that NSGA-II does not produce a single solution as GA, but a set of solutions. As shown in Figure 6, NSGA-II converges towards Pareto-optimal solutions that are considered as good compromises between quality and semantic coherence. In this figure, each point is a solution with the quality score represented in x-axis and the semantic coherence score in the y-axis. The best solutions exist in the right-bottom corner representing the Pareto-front that maximizes the semantic coherence value and maximizes the quality. The maintainer can choose a solution from this front depending on his preferences in terms of compromise. However, at least for our evaluation, we need to select only one solution. To this end and in order to fully automate our approach, we propose to extract and suggest only one best solution from the returned set of solutions. Equation 11 is used to choose the solution that corresponds of the best compromise between *Quality* and *Semantic Coherence*. In our case the ideal solution has the best quality value (equals to 1) and the best semantic coherence value (equals to 1). Hence, we select the nearest solution to the ideal one in terms of Euclidian distance

$$bestSol = \underset{i=0}{\overset{n-1}{Min}}\left(\sqrt{(1-Quality[i])^2 + (1-Semantic[i])^2}\right) \quad (11)$$

where n is the number of solutions in the Pareto front returned by NSGA-II.

As shown in Figure 6, since the two objectives of quality and semantic coherence are conflicting, the results confirm that a solution which scores better in code-quality is better, in terms of quality, than any other solution which is of lower semantic coherence.

The correction results might vary depending on search space exploration, since solutions are randomly generated, though guided by a meta-heuristic. To ensure that our results are relatively stable, we compared the results of multiple executions for NSGA-II as shown in Figure 7. We, consequently, believe that our technique is stable, since the semantics and quality scores are approximately the same for three different executions. In the three different executions, we obtained better results in terms of semantic coherence compared to the two other approaches.

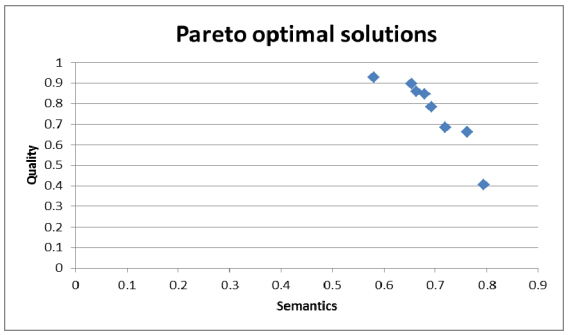

Fig 6. An example of Pareto optimal solutions for Xerces-J

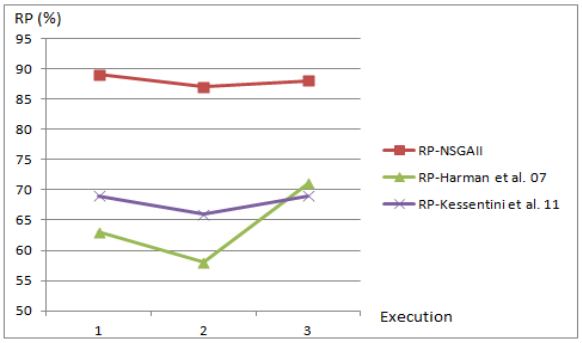

Fig 7. An example of multiple executions on Xerces-J

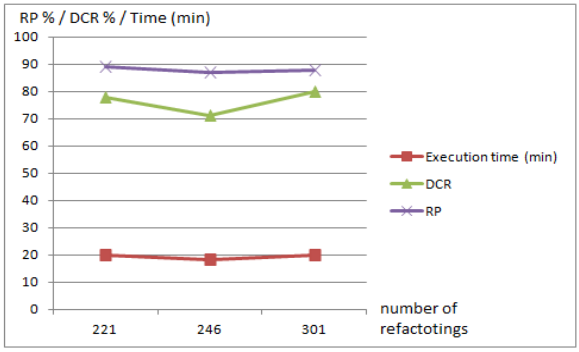

Fig 8. Number of refactoring impact

Finally, it is important to contrast the results with the execution time. The execution time for finding the optimal refactoring solution with a number of iterations (stopping criteria) fixed to 1000 was less than twenty minutes as shown in Figure 8. This indicates that our approach is reasonably scalable from the performance standpoint. In addition, we evaluate the impact of the number of suggested refactorings on the RP and DCR scores in three different executions. The best RP and DCR scores are obtained with the higher number of suggested refactorings. Thus, we could conclude that our approach is scalable and the results accuracy is not affected by the number of suggested refactoirngs.

VI. Related Work

Several studies have been focused on software refactoring in recent years. In this section, we summarize existing approaches where search-based techniques have been used to automate refactoring activities. We classify the related work into two main categories: mono-objective and multi-objective optimization approaches. Then, we describe some work that use semantic and conceptual dependencies in the context of impact analysis and test cases generation.

In the first category, the majority of existing works combine several metrics in a single fitness function to find the best sequence of refactorings. Seng et al. [4] have proposed a single-objective optimization based on genetic algorithm (GA) to suggest a list of refactorings. The search process uses a single fitness function to maximize a weighted sum of several quality metrics. Used metrics are related to some properties such as coupling, cohesion, complexity and stability. Indeed, the authors used some preconditions for each refactoring. These conditions are able to preserve the program behavior (refactoring feasibility), but not the domain semantics. In addition, the validation was done only on the move method refactoring. O'Keeffe et al. [11] have used different local search-based techniques such as hill climbing and simulated annealing, to provide automated refactoring support. Eleven weighted object-oriented design metrics have been used to evaluate the quality improvements. In [14], Qayum et al. considered the problem of refactoring scheduling as a graph transformation problem. They expressed refactorings as a search for an optimal path, using Ant colony optimization, in the graph where nodes and edges represent respectively refactoring candidates and dependencies between them. However the use of graphs does not consider the domain semantics of the program and its runtime behavior. Fatiregun et al. [26] have proposed a search-based approach for finding program transformations. They apply a number of simple atomic transformation rules called axioms. They presume that if each axiom preserves semantics then a whole sequence of axioms ought to preserve semantics equivalence. However, this is context-dependent and therefore not always valid. Indeed, the semantic equivalence is based only on structural rules and no semantic code analysis have been proposed.

In [1], Kessentini et al. have proposed a single-objective combinatorial optimization using a GA to find the best sequence of refactoring operations that improve the quality of the code by minimizing the number of design defects detected. However, they do not preserve the domain semantics of the initial design when applying refactorings.

In the second category, Harman et al. [3] have proposed a multi-objective approach using Pareto optimality that combines two quality metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class), in two separate fitness functions. The multi-objective algorithm could find a good sequence of move method refactorings that should provide the best compromise between CBO and SDMPC to improve code quality. However, in addition to restrict the solutions to only move method operations, the domain semantics is not preserved. In [2], we proposed a multi-objective optimization to find

the best sequence of refactorings using NSGA-II. This approach is based on two fitness functions, quality (proportion of corrected defects) and code modification effort, to recommend a sequence of refactorings that provide the best tradeoff between quality and effort. To conclude, the vast majority of existing search-based approaches focused only on the program structure improvements without taking into consideration the domain semantics.

Other contributions are based on rules that can be expressed as assertions (invariants, pre and post-condition). The use of invariants has been proposed to detect parts of program that require refactoring by [13]. Opdyke [12] propose the definition and the use of pre- and post-condition with invariants to preserve the behavior of the software. Behavior preservation is based on the post-condition verification. All these conditions could be expressed in terms of rules.

There exists some research work that investigates semantics and conceptual dependencies. Steimann et al. [16] take into consideration the semantics preservation when generating program mutants. A constraint-based refactoring approach is proposed to generate mutant programs that are syntactically and semantically correct. In [18], a technique inspired from information retrieval aims to identify conceptual dependencies to analyze the code impact change. In [19], a similar approach based on "relational topic model" is proposed to capture dependent topics in classes and relationships among them for impact analysis. Furthermore, Zhang et al. [17] uses static analysis to identify method dependencies relations to guide automated test cases generation based on shared fields.

VII. CONCLUSION

This paper presents a novel search-based approach that suggests "good" sequence of refactoring operations to correct detected design defects. The suggested refactorings preserve the domain semantics of the program to restructure while correcting existing design defects. Our search-based approach succeeded to produce more semantic and meaningful refactorings in comparison of those of the state of the art. The proposed approach was tested on two open-source systems, and compared successfully to two existing approaches.

As part of future work, we plan to make a comparative study with different existing techniques and extend our experiments on other badly-designed projects using more programming contexts.

Acknowledgement. This publication was made possible by NPRP grant # [09-1205-2-470] from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the author[s].

REFERENCES

[1] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, A. Ouni, Design Defects Detection and Correction by Example, 19th Int. Conf. on Program Comprehension (ICPC), pp. 81-90, 2011.

[2] A. Ouni, M. Kessentini, H. Sahraoui and M. Boukadoum, Maintainability Defects Detection and Correction: A Multi-Objective Approach. J. of Autmated Software Engineering, Springer, 2012.

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

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

[5] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring—Improving Coupling and Cohesion of Existing Code," Proc. 11th Working Conf. on Reverse Eng. pp. 144-151, 2004.

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

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

[8] E. Zitzler, and L. Thiele, Multiobjective optimization using evolutionary algorithms—A comparative case study. In Parallel Problem Solving from Nature, pp.292–301, Springer, Germany, 1998.

[9] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput., vol. 6, pp. 182–197, Apr. 2002.

[10] http://www.refactoring.com/catalog/

[11] M. O'Keeffe, and M. O. Cinnéide, Search-based Refactoring for Software Maintenance. J. of Systems and Software, 81(4), 502–516.

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

[13] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin, Automated support for program refactoring using invariants, in Int. Conf. on Software Maintenance (ICSM), pp. 736–743, 2001.

[14] F. Qayum, R. Heckel, Local search-based refactoring as graph transformation. Proceedings of 1st International Symposium on Search Based Software Engineering; pp. 43–46, 2009.

[15] R. Heckel, Algebraic graph transformations with application conditions, M.S. thesis, TU Berlin, 1995.

[16] F. Steimann and A. Thies. From behaviour preservation to behaviour modification: constraint-based mutant generation. In Int. Conf. on Software Engineering (ICSE), pp. 425–434, 2010.

[17] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In Int. Symp. on Software Testing and Analysis (ISSTA), pp. 353–363, 2011.

[18] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In Work. Conf. on Reverse Engineering (WCRE), pp. 119–128, 2010.

[19] M. Gethers and D. Poshyvanyk. Using relational topic models to capture coupling among classes in object-oriented software systems, Int. Conf. on Software Maintenance (ICSM), 2010.

[20] T. Mens, T. Tourwé: A Survey of Software Refactoring. IEEE Trans. Software Eng. 30(2), pp. 126-139, 2004.

[21] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, V. Sundaresan, Optimizing Java bytecode using the Soot framework: Is it feasible? in Int. Conf. on Compiler Construction, pp. 18–34, 2000.

[22] R. B. Yates and B. R. Neto. Modern Information Retrieval, ADDISON-WESLEY, New York, 1999.

[23] http://www.eclipse.org/

[24] http://ganttproject.biz/index.php

[25] http://xerces.apache.org/

[26] D. Fatiregun, M. Harman, and R. M. Hierons. Evolving transformation sequences using genetic algorithms. In Proc. of 4th SCAM, pages 66-75, 2004.