Search-Based Refactoring Using Recorded Code Changes

Conference Paper *in* Proceedings of the Euromicro Conference on Software Maintenance and Reengineering, CSMR · March 2013

DOI: 10.1109/CSMR.2013.31

Ali Ouni
Osaka University
39 PUBLICATIONS **356** CITATIONS

2013 17th European Conference on Software Maintenance and Reengineering

Search-based Refactoring Using Recorded Code Changes

Ali Ouni[1,2], Marouane Kessentini[2], and Houari Sahraoui[1]
[1]DIRO, Université de Montréal, Canada
{ouniali, sahraouh}@iro.umontreal.ca
[2]CS, Missouri University of Science and Technology, USA
marouanek@mst.edu

Abstract— **Over the past decades, many techniques and tools have been developed to record the sequence of applied refactoring to improve design quality. We start from the observation that these recorded code changes can be used to propose new refactoring solutions in similar contexts. In addition, this knowledge can be combined with structural and semantic information, used by existing work, to improve the automation of refactoring. In this paper, we propose a multi-objective optimization approach to find the best sequence of refactorings that maximizes the use of refactoring applied in the past to similar contexts, minimizes semantic errors and minimizes the number of defects (improve code quality). To this end, we use the non-dominated sorting genetic algorithm (NSGA-II) to find the best trade-off between these three objectives. We report the results of our experiments on different open source java projects.**

Keywords- Search-based Software Engineering; Refactoring; Software Maintenance; Multi-objective Optimization

I. INTRODUCTION

Source code of large systems is iteratively refined, restructured and evolved due to many reasons such as correcting errors in design; modifying a design to accommodate changes in requirements; and modifying a design to enhance existing features. To ease these maintenance activities, one of the most-used techniques is refactoring which improves design structure while preserving the external behavior [13] [16]. Roughly speaking, we can identify two distinct steps in the refactoring process: (1) detect when an application should be refactored and (2) identify which refactorings should be applied and where. In this paper, we focus on the latter.

Various tools supporting refactoring have been proposed in the literature [20] [32]. The vast majority of these tools provides different environments to apply manually or automatically refactoring operations to fix, for examples, bad smells [8]. In addition, several remarkable techniques have been proposed to detect the changes applied to a model by either recording the changes directly in the modeling editor as they are applied by a developer [24] [25] [27] [28] [30] or by analyzing two versions of a model and computing the differences between them. In general, these approaches detect changes between two (or more) code versions by composing atomic changes to refactoring operations. However, in most of the existing work the recorded refactoring between different versions are only used to help the maintainer to understand code evolution.

The majority of existing contributions does not use recorded refactoring to propose new ones in similar contexts. These contributions formulate the refactoring problem as a single-objective optimization problem that maximizes code quality while preserving the behavior [12] [3] [5] [1]. In some other work, other objectives are also evaluated such as reducing the effort (number of code changes) [2], semantic preservation [4], and quality metrics improvements [3]. However, sometimes structural and semantic information is not enough to generate efficient refactoring [4].

We start from the observation that good recorded code changes can be used to propose new refactoring solutions in similar contexts. This knowledge can be combined with structural and semantic information, used by existing work, to improve the automation of refactoring. In this paper, our aim is to improve the quality of suggested refactoring solutions that are generated using our search based approach, by maximizing the use of refactoring applied in the past (recorded refactoring) to similar contexts. For instance, if a *move method getName()* is applied/recorded in the past between *class Student and Person* thus this will increase the probability that a new *move method* between these two classes can make sense (preserving the semantic).

We propose a multi-objective optimization approach to find the best sequence of refactorings that maximizes the use of refactoring applied in the past to similar contexts, minimizes semantic errors and minimizes the number of defects (improve code quality). The proposed algorithm is an adaptation of the Non-dominated Sorting Genetic Algorithm (NSGA-II) [10], a recent multi-objective evolutionary algorithm (MOEA) [9]. NSGA-II aims at finding a set of representative Pareto optimal solutions in a single run. The evaluation of these solutions is based on the three mentioned criteria. We evaluate first if similar changes are applied in previous versions of the code fragments that will be refactored by the generated solution. Then, based on our earlier work, we use a combination of quality metrics as an indication of the presence of bad-smells and two techniques are combined to estimate the semantics proximity between classes when moving elements between them (dependencies between classes extracted from call graphs and vocabulary similarity).

The rest of this paper is organized as follows. Section 2 is dedicated to the problem statement, while Section 3

1534-5351/13 $26.00 © 2013 IEEE
DOI 10.1109/CSMR.2013.31

describes our approach details. Section 4 explains the experimental method, the results of which are discussed in Section 5. Section 6 introduces related work, and the paper concludes with Section 7.

II. Problem Statement

In this section, we detail the specific issues that are addressed by our approach to improve the automation of refactoring using recorded code changes. Furthermore, we describe a motivating example that illustrates some of these issues.

Refactoring is the process of improving code design quality without altering the external behaviour. As defined by [16], refactoring includes different steps: (1) the identification of refactoring opportunities, for example, by detecting bad-smells [7] [8], (2) the selection of refactoring operations that can be used to improve the design quality (e.g. fixing bad-smells), and (3) the verification if applied refactorings preserve the behaviour. In this work, a variety of refactoring operation types is used such as *move method*, *extract class*, *move field*, etc. More details of our used refactorings can be found in [11].

Various techniques are proposed to automate the refactoring process [5] [2] [3] [1] [12]. Most of these techniques are based on structural information using a set of quality metrics. The structural information is used to ensure that applied refactorings improve some quality metrics such as the number of methods/attributes per class. However, this is not enough to confirm that a refactoring makes sense and preserves the design semantic coherence. We need to preserve the rationale behind why and how code elements are grouped and connected.

To solve this issue, semantic measures are used to evaluate refactoring suggestions such as those based on coupling and cohesion or information retrieval techniques (e.g. cosine similarity of the used vocabulary) [4]. However, these semantic measures depend heavily on the meaning of code elements name/identifier (e.g., name of methods, name of classes, etc.). Indeed, due to some time-constraints, developers select meaningless names for classes, methods, or fields (not clearly related to the functionalities) [33]. Thus, it is risky to use or adapt only techniques such as cosine similarity to find a semantic approximation between code fragments. In addition, when applying a refactoring like move method between two classes many target classes can have the same values of coupling and cohesion with the source class. To make the situation worse, it is also possible that the different target classes have the same structure.

Figure 1 illustrates an example, extracted from the open-source java project JHotDraw v5.2 [20]. We consider a design fragment containing three classes *ArrowTip*, *AbstractLineDecoration* and *AttributeFigure*. We propose the scenario that a refactoring solution consists to *move a method* from the class *ArrowTip* to another target class. Based on structural and semantic information the two possible target classes can be *AbstractLineDecoration* and *AttributeFigure*. However, these classes have approximately the same structure that can be formalized using quality metrics (e.g. number of methods, number of attributes, etc.) and their semantics similarity is close to *ArrowTip* using a vocabulary-based measure, or cohesion and coupling [...]. In this case, structural and semantics information is not sufficient to decide about the best refactoring solution. But, from previous versions of JHotDraw, we recorded that the method *draw()* has been moved from the class *ArrowTip* to the class *AbstractLineDecoration*. As a consequence, moving methods and/or attributes from the class *ArrowTip* to the class *AbstractLineDecoration* has higher correctness probability than moving methods or attributes to the class *AttributeFigure*. We use, then, this observation that recorded refactoring can be used to suggest new ones in similar contexts as our working hypothesis in this paper.

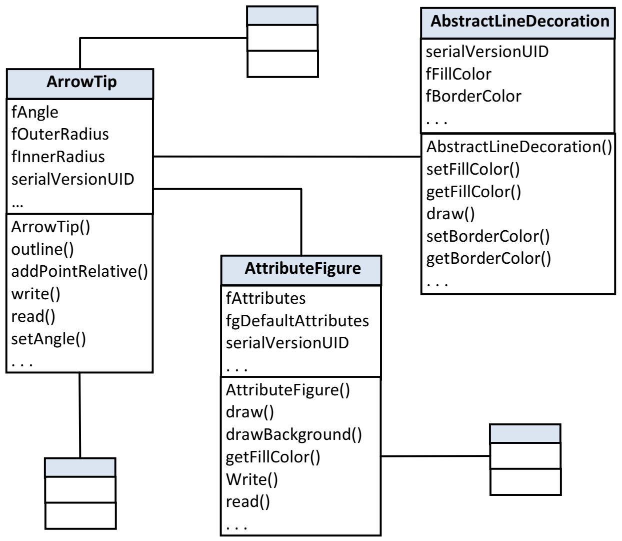


Fig 1. Design fragment extracted from JHotDraw v5.2

Several remarkable techniques have been proposed for recording code changes directly in the development framework as they are applied by a developer [28] [30] or by analyzing two versions of a system and computing the differences between them [25]. The use of recorded/collected refactoring is at origin of our work that will be described in the next sections.

III. Refactoring Using Multi-Objective Optimization

In this section, we summarize our previous work about multi-objective for code refactoring where quality and semantic coherence are used as optimization objectives. Then, we give an overview about our proposal where an additional objective is added to overcome the different limitations discussed in Section 2.

A. Refactroing as a Multi-Objective Problem

In our previous work [4] we proposed an automated approach, to ameliorate the quality of a system while preserving its domain semantics. It uses multi-objective optimization to find the best compromise between code quality improvements and domain semantics preservation. Improving software quality corresponds to correcting design defects. We used defects detection rules, proposed in our previous work [2], to find the best refactoring solutions, from an exhaustive list of refactorings, which minimizes the number of detected defects. Preserving the semantic coherence after applying the suggested refactorings is ensured by maximizing different semantic measures: 1) vocabulary-based similarity (cosine similarity between words: name of code elements; and 2) dependency-based similarity (call-graph, coupling and cohesion).

Due to the large number of possible refactoring solutions [11] 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 [4]. 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]. The process of generating a correction solution (refactorings) 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.

Due to the different limitations described in Section 2, we extend, in this paper, our previous work by considering the history of applied refactoring in addition to structural and semantic information.

B. *Approach Overview*

The general structure of our approach is introduced in Figure 2. It takes as inputs a history of applied code changes/refactoring to previous versions of the system (label B), an exhaustive list of possible refactoring that can be applied (label A), a set of bad-smells detection rules (label C) [2], and some semantic measures used in our previous work (label D) [4]. Our approach generates as output the best sequence of refactoring, selected from the exhaustive list, that maximizes the use of refactoring applied in the past to similar contexts, minimizes semantic incoherence and minimizes the number of defects (improve code quality).

The history of applied refactoring in the past can be collected using two different techniques. The first technique is to capture applied refactorings by tracking their execution in the development environment directly [28]. In contrast to refactoring tracking approaches, state-based refactoring detection mechanisms aim to reveal refactorings a posteriori on the base of the two successively modified versions of a software artifact [30]. Then, we use this history of collected refactoring to evaluate the application of new changes. To find the best trade-off between the three objectives (quality, semantic coherence and similarity with refactoring history), we adapted the non-dominated sorting genetic algorithm (NSGA-II) [10]. This algorithm and its adaptation to the refactoring problem are described in the next section.

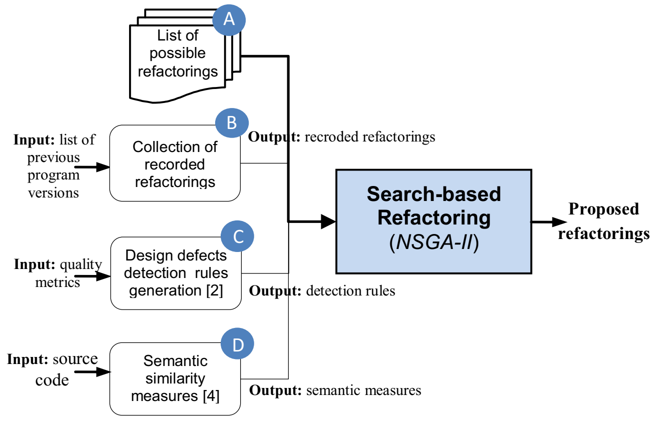

List of possible refactorings
A
Input: list of previous program versions
Collection of recorded refactorings
B
Output: recroded refactorings
Search-based Refactoring (NSGA-II)
Proposed refactorings
Input: quality metrics
Design defects detection rules generation [2]
C
Output: detection rules
Input: source code
Semantic similarity measures [4]
D
Output: semantic measures

Fig 2. Multi-objective refactoring using recorded code changes

IV. Similarity with Recorded Code Changes as a New Objective

This section describes the principles of our approach for improving refactoring suggestions using past recorded code changes. It starts by presenting an overview about the used NSGA-II algorithm. Then, we provide the details on how similarity with recorded code changes is considered as a new objective in NSGA-II. The motivating example described in Section 2 is used to illustrate our approach described in the rest of this paper.

A. *NSGA-II adaptation*

NSGA-II is a powerful search method stimulated by natural selection that is inspired from the theory of Darwin [10]. 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, so-called non-dominated solutions, or also Pareto set. A non-dominated solution 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.

We describe in this section how we adapted the NSGA-II algorithm to find the best tradeoff between quality, semantic coherence and similarity with recorded refactoring. As our aim is to maximize the quality, minimize semantic errors, and maximize similarity with a refactoring history, we consider each one of these criteria as a separate objective for NSGA-II.

As shown in Figure 3, NSGA-II starts by generating an initial population based on a specific representation that will be discussed later, using the exhaustive list of refactoring operations given at the input, as mentioned in Section III.B. Thus, this population stands for a set of possible bad smells-correction solutions represented as sequences of refactoring which are randomly selected and combined.

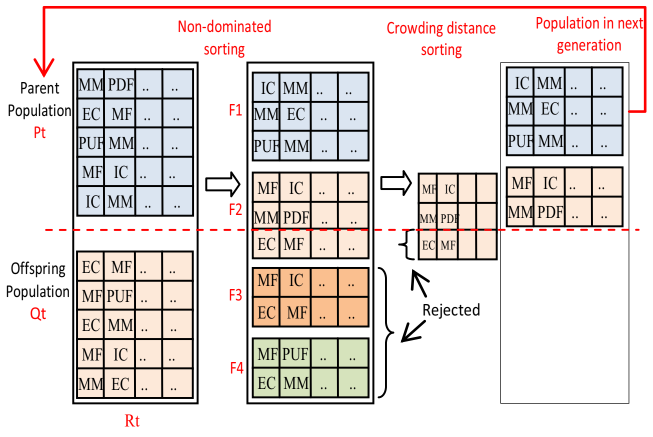


Fig 3. NSGA-II overview

After generating a population of refactoring solutions, the main NSGA-II loop goal is to make a population of candidate solutions evolve toward the best sequence of refactoring, *i.e.,* the sequence that minimizes as much as possible the number of bad smells, preserves the domain semantics and maximizes the similarity with recorded refactoring. During each iteration t, an offspring population Q_t is generated from a parent population P_t using genetic operators (selection, crossover and mutation). Then, Q_t and P_t are assembled in order to create a global population R_t. Then, each solution S_i in the population R_t is evaluated using our three fitness functions: (1) *quality function* to maximize: represents the number of detected bad smells after applying the generated refactoring sequence, (2) *semantic incoherence function* to minimize: calculates the semantics similarity of the elements to be changed by the refactoring using the semantic measures described in the previous section, and (3) *similarity with past/recorded refactoring function* to maximize: consists to evaluate the similarity between proposed refactoring types and those applied in the past to the same code fragments.

Once these functions are calculated, all solutions will be sorted in order to return a list of non-dominated fronts F (F_1, F_2, ...), where F_1 is the set of non-dominated solutions, F_2 is the set of solutions dominated only by solutions in F_1, etc. When the whole current population is sorted, the next population P_{t+1} will be created using the half top-ranked solutions, starting from front F_1. When the half is reached inside a front F_i, solutions of F_i, with the same dominance, are sorted using a normalized average of the three objectives. After that, as mentioned earlier, genetic operators are applied to produce the set of solutions Q_{t+1} to complete the population R_{t+1}. The algorithm terminates when it achieves the termination criterion (maximum iteration number). The output of the algorithm is the set of best solutions, *i.e.*, those in the Pareto front of the last iteration.

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

1) Solution Representation

In this paper, a variety of refactoring operations are considered. Each of these refactoring operations takes a list of controlling parameters as illustrated in Table 1. For example, the operation *move method(sourceClass, targetClass, method)* indicates that a method is moved from a source class to a target class.

Table 1. Refactoring operations with its controlling parameters.

Ref	Refactorings	Controlling parameters
MM	Move Method	(sourceClass, targetClass, method)
MF	Move Field	(sourceClass, targetClass, field)
PUF	Pull Up Field	(sourceClass, targetClass, field)
PUM	Pull Up Method	(sourceClass, targetClass, method)
PDF	Push Down Field	(sourceClass, targetClass, field)
PDM	Push Down Method	(sourceClass, targetClass, method)
IC	Inline Class	(sourceClass, targetClass)
IM	Inline Method	(sourceClass, sourceMethod, targetClass, targetMethod)
EM	Extract Method	(sourceClass, sourceMethod, targetClass, extractedMethod)
EC	Extract Class	(sourceClass, newClass)

A vector is used to represent a candidate solution. Each dimension represents a refactoring operation to apply. The refactoring operations of a solution have to be applied in the order of their positions in the vector. Figure 4 describes an example of a population containing 8 individuals. Two of these solutions are illustrated in details containing respectively 6 and 5 operations. The same refactoring type can appear many times in one solution but with different controlling parameters. However, for each of these refactorings, we specified pre- and post-conditions that are already studied in [7] to ensure the feasibility of applying them. These constraints are also used when genetic operators are applied.

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 refactorings that are considered and the size of the program to be refactored. A higher number of operations in a solution does 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 given refactoring list. During the creation, the solutions have random sizes inside the allowed range. In this next subsection, we define how a new population is generated from an existing one using the change operators.

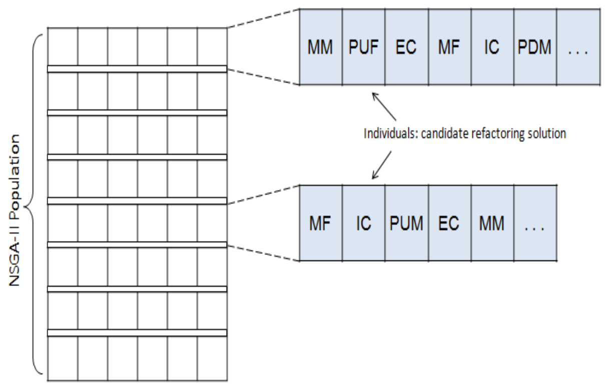


Fig 4. Population generation and individual representation

2) *Change operators*

The selection of individuals that will undergo the crossover and mutation operators in our adaption of NSGA-II is based on the Stochastic Universal Sampling algorithm (SUS). SUS is a random selection algorithm which gives higher probability to be selected to the fittest solutions while still giving a chance to every solution.

When two-parent individuals are selected, and a random cut point is determined to split them into two sub-vectors. Then, crossover swaps the sub-vectors from one parent to the other. Thus, each child combines information from both parents. This operator must enforce the length limit constraint by eliminating randomly some refactoring operations.

Figure 5 shows an example of the crossover process. In this example, Parent 1 and Parent 2 are combined to generate two new solutions. The right sub-vector of Parent 1 is combined with the left sub-vector of Parent 2 to form the first child, and the right sub-vector of Parent 2 is combined with the left sub-vector of Parent 1 to form the second child.

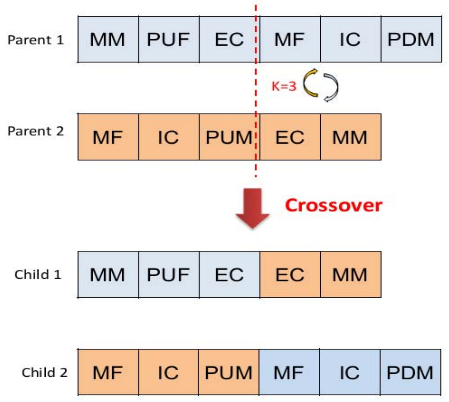


Fig 5. Crossover operator

The mutation operator starts by randomly selecting one or more dimensions (refactoring operation) from the solutions (vector). Then, the selected operation(s) are replaced by other ones from the initial list of in the exhaustive refactoring set. An example is shown in figure 6 where two dimensions are replaced by new refactoring. In addition, the mutation can only modify the controlling elements of some dimensions without replacing the operation by a new one.

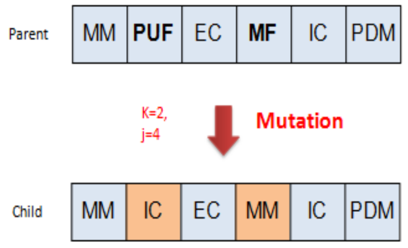


Fig 6. Mutation operator

3) *Multi-criteria evaluation (Fitness Functions)*

A maximum of two different objectives are used in existing work related to search-based refactoring [2] [4]. In this work, we are using three different fitness functions to include in our NSGA-II adaptation.

a) *Improving design quality (structural information)*

The Quality criterion is evaluated using the fitness function given in Equation (1). 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 bad smells detection rules) 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].

$$Quality = \frac{\#\, corrected\ defects}{\#\, defects\ before\ applying\ refactorings} \qquad (1)$$

b) Preserving semantic coherence

The semantics Sem(RO_i) of a refactoring operation RO_i corresponds to the weighted sum of each measure (vocabulary and dependency-based similarity) [4] used to approximate the semantic 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 as illustrated in Equation (2).

$$Semantic = \sum_{n=0}^{n-1} Sem(RO_i)/n \qquad (2)$$

where n is the size of the solution, i.e., the number of refactoring operations. More details can be found in our prior work [4].

c) Maximizing similarity with a refactoring history

The overall idea is to maximize/encourage the use of new refactoring that are similar to those applied to same code fragments in the past. To calculate the similarity score between a proposed refactoring operation and different recorded/collected code changes, we use the following fitness function:

$$Sim\_refactoring\_history(ROi) = \sum_{j=0}^{n-1} w * m \qquad (3)$$

where n is the size of the list of possible refactoring operations we use, m is the number of times that the same refactoring type has been applied in the past to the same code fragment/element and w is a refactoring weight that calculates the similarities between refactoring types if an exact matching cannot be found with the base of recorded refactoring.

Table 2 explains how the refactoring similarity weights are calculated. We consider two refactorings as similar if one of them is partially composed by the other or if their implementations are similar (using equivalent controlling parameters described in Table 1). Indeed, some complex refactorings such as extract class can be composed by move method, move field, create a new class, etc. In general, the weight w takes the values: 2 if the refactoring is the same (for example, many move methods can be applied between the same source and target classes) or have similar implementation, 1 if one of the refactoring is compatible or composed partially by the other otherwise 0 if the two refactorings are completely different.

To calculate the similarity score of the whole proposed refactoring solution with historical code changes, we calculate the sum of the similarity value of each refactoring operation in this solution.

Table 2. Similarity weights between refactoring types.

	MM	MF	PUF	PUM	PDF	PDM	IM	EM	EC
MM	2	2	1	1	1	1	1	2	2
MF	2	2	1	1	1	1	2	2	2
PUF	1	2	2	2	1	1	0	0	2
PUM	1	1	2	2	1	1	0	0	2
PDF	1	1	1	1	2	2	0	0	2
PDM	2	1	1	1	2	2	0	0	2
IM	1	0	0	1	0	1	2	0	1
EM	1	1	1	1	1	1	0	1	2
EC	1	1	1	1	1	1	0	0	1

V. VALIDATION

In order to evaluate the feasibility of our approach for generating good refactoring suggestions, we conducted an experiment based on different versions of large open source systems. We start by presenting our research questions. Then, we describe and discuss the obtained results. All the experimentation materials can be found in [26].

A. Objectives

In our study, we assess the performance of our refactoring proposal by finding out whether it could generate meaningful sequences of refactorings to correct bad-smells, preserve the semantics of the design, and reuse as much as possible a base of recorded code changes. Indeed, our study addresses two research questions, which are defined below. We also explain how our experiments are designed to address them. The two research questions are:

RQ1: To what extent can the reuse of recoded refactorings improves the results of refactoring suggestion compared to work that do not use them?

RQ2: To what extent the proposed multi-objective approach performs better compared to a random search or a mono-objective approach?

To answer these questions, 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. Finally, we compare our results, using these measures, to those produced by some of the existing work where one or two objectives are used (Quality and/or semantics). For instance, in [1] we used a single-objective genetic algorithm to correct defects by finding the best refactoring sequence that reduces the number of defects. In another work [4], we considered also the semantic preservation. We evaluate if our refactoring solutions, using three objectives, produces less incoherent changes than those proposed by [1] and [4] while having similar quality improvement. We also asses the performance of the multi-objective algorithm NSGA-II compared to a random search, and a genetic algorithm where one fitness function is used (an average of the three objective scores).

B. Setting

We used two open-source Java projects to perform our experiments: GanttProject (Gantt for short) v1.10.2 [21], JHotDraw v6.0b1 [20][23], and Xerces-J v2.7.0 [22]. Table 3 provides some descriptive statistics about these two programs.

Table 3. Open-source systems statistic.

Systems	Number of classes	KLOC	Number of defects
GanttProject v1.10.2	245	31	41
Xerces-J v2.7.0	991	240	66
JHotDraw v6.0b1	585	23	21

We choose Xerces-J, JHotDraw 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 and JHotDraw have been actively developed over the past 10 years, and their design has not been responsible for a slowdown of their developments. Table 4 describes the different versions used and the number of collected/extracted refactoring between these different versions. To ensure the similarity with good refactorings, we manually inspected a huge list of collected refactorings, only good refactorings are selected and used in our approach.

Table 4. Open-source systems versions and recorded/collected refactoring.

Systems	Used versions for collecting refactorings	Number of collected/ recorded refactorings
GanttProject v1.10.2	GanttProject v1.7, GanttProject v1.8, GanttProject v1.9.10	58
Xerces-J v2.7.0	Xerces-J v1.4.2, Xerces-J v2.5.0, Xerces-J v2.6.0, Xerces-J v2.6.1	70
JHotDraw v6.0b1	JHotDraw v5.1, JHotDraw v5.2, JHotDraw v5.3	64

To answer our two research questions, we derived three metrics: defect correction ratio (DCR), refactoring precision (RP), and reused refactoring (RR). DCR is given by (4) and calculates the number of corrected defects over the total number of defects detected before applying the proposed refactoring sequence.

$$DCR = \frac{\#\,corrected\ defects}{\#\,defects\ before\ applying\ refactorings} \quad (4)$$

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 [19] 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 (5)

$$RP = \frac{\#\,meaningful\ refactorings}{\#\,proposed\ refactorings} \quad (5)$$

In order to evaluate the importance of reusing collected refactorings in similar contexts, we defined a third metric, reused refactoring (RR), that calculates the percentage of operations from the base of collected changes used to generate the optimal refactoring solution using our proposal.

$$RR = \frac{\#\,used\ refactorings\ in\ the\ base\ of\ code\ changes}{\#\,refactorings\ in\ the\ base\ of\ code\ changes} \quad (6)$$

C. Results and Discussions

Table 5 summarizes our findings. The majority of suggested refactorings improve significantly the code quality while preserving the semantics much better than two other approaches. The results show that half of recorded/collected refactorings are used by the optimal solution. We found also that the best compromise is obtained between the three objectives using NSGA-II comparing to the use of only quality and/or semantics.

For Gantt, 178 of the 187 proposed refactoring operations (95%) make sense semantically and correct 87% of defects. This score is much better than the two other approaches, with quality and/or semantics objectives, having respectively 91% and 73% as RP scores and correct the same number of bad smells (87%). The corrected defects were from 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. In addition, the combination of quality and collected refactoring history are lower than our 3-objectives results in terms of semantics preservation and quality improvement. Thus, the use of collected refactoring history reduces the number of semantic incoherence when applying refactoring operations. Since the defect correction scores are very close, we could sacrifice a small quality decrease to improve the semantic coherence of the program design by reusing a refactoring history. Since Gantt versions include a good distribution of code changes, almost half of the refactoring history is used by the multi-objective algorithm to generate the optimal solution.

We had similar results for Xerces-J and JHotDraw. The majority, at least 77%, of defects was corrected and most of the proposed refactoring sequences, at least 92%, are coherent semantically. When comparing with previous work, the use of up-to two objectives perform slightly better for the number of corrected defects than NSGA-II. For the refactoring precision, strategies proposed by NSGA-II require less manual adaptation than those of Kessentini et al. [1]. Indeed, the majority of refactoring operations proposed by NSGA-II do not need programmer intervention to solve semantic incoherences that are introduced when applying them. The reused percentage of refactoring history (RR) is slightly lower than Gantt but still acceptable (more than 38%). This is can be explained by the reason that Xerces and JHotDraw have a huge list of operations recorded/collected in the past than Gantt (Table 4).

Table 5. Refactoring results for NSGA-II.

Systems	Objectives			DCR	RP	RR
	Quality	Semantics	Refactoring Reuse			
GanttProject v1.10.2	x			95% (39|41)	73% (158|216)	N.A
	x	x		87% (36|41)	91% (128|146)	N.A
	x		x	93% (36|41)	81% (159|194)	67%
	x	x	x	87% (36|41)	95% (178|187)	43%
Xerces-J v2.7.0	x			89% (59|66)	69% (212|304)	N.A
	x	x		77% (51|66)	89% (197|221)	N.A
	x		x	89% (59|66)	78% (220|281)	62%
	x	x	x	77% (51|66)	93% (219|236)	41%
JHotDraw v6.0b1	x			90% (19|21)	68% (146|213)	N.A
	x	x		81% (17|21)	87% (139|160)	N.A
	x		x	86% (18|21)	83% (162|195)	56%
	x	x	x	81% (17|21)	92% (173|188)	38%

Another element that should be considered when comparing the results of the different mono/multi-objective algorithms, is that NSGA-II does not produce a single solution as GA, but a set of solutions. As shown in Figure 7, NSGA-II converges to Pareto-optimal solutions that are considered as good compromises between quality, semantic coherence, and reusing refactoring history. In this figure, each point is a solution with the quality score represented in x-axis, the semantic coherence score in the y-axis and the refactoring reuse in z-axis. The best solutions exist in the corner representing the Pareto-front that maximizes the semantic coherence value, refactoring reuse and the quality. The maintainer can choose a solution from this front depending on his preferences in terms of compromise. However, at least for our validation, we need to have only one best solution that will be suggested by our approach. To this end and in order to fully automate our approach, we propose to extract and suggest only one from the returned set of best solutions. Equation 7 is used to choose the solution that corresponds of the best compromise between *Quality, Semantic Coherence,* and *Refactoring Reuse.* In our case the ideal solution has the best quality value (equals to 1), the best semantic coherence value (equals to 1) and the maximum number of reused refactoring. Hence, we select the nearest solution to the ideal one in terms of Euclidian distance:

$$bestSol = \underset{i=0}{\overset{n-1}{Min}}\left(\sqrt{(1-\mathrm{Quality}[i])^2+(1-\mathrm{Semantic}[i])^2+(1-\mathrm{SimRefHist}[i]/\mathrm{max\_Hist})^2}\right) \quad (7)$$

where n is the number of solutions in the Pareto front returned by NSGA-II, and *max_Hist* is the maximal similarity value with refactorings history in the Pareto front.

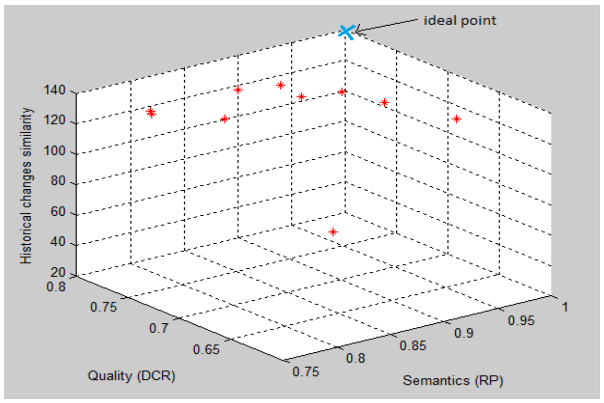
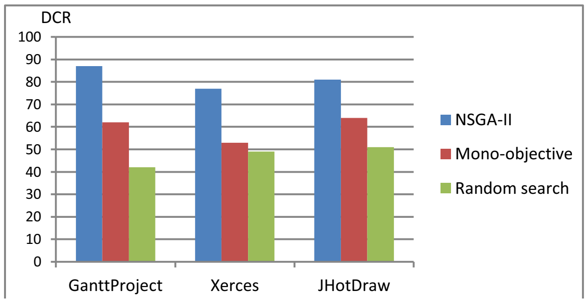
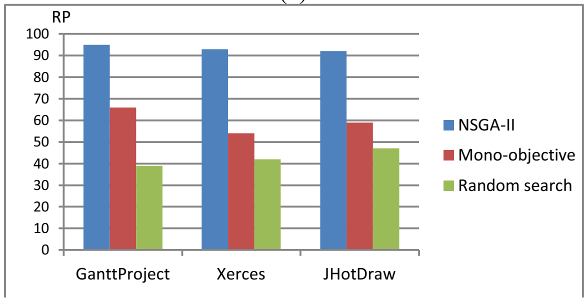


Fig 7. Example of Pareto front for JHotDraw 6.0b1

To evaluate the efficiency of NSGA-II and justify the need to use multi-objective algorithms, we compared the performance of our proposal to a random search and mono-objective algorithms. In a random search, the change operators (crossover and mutations) are not used, and populations are generated randomly and evaluated using the three fitness functions. In our mono-objective adaptation, we considered only one fitness function which is the average score of the three objectives. Since in our NSGA-II adaptation we select a single point without giving more we (equal weights). As shown by Figure 8, NSGA-II outperforms significantly comparing to random-search and mono-objective algorithms in terms of corrected bad-smells and semantic preservation. For instance, in Gantt-Project NSGA-II has approximately the double of random/mono-objective RP and DCR scores.



(a)



(b)

Fig 8. Comparison between random search, mono-objective, and multi-objective algorithms for refactoring suggestions: (a) Quality (DCR) and (b) Semantics preservation (RP).

Finally, the execution time for finding the optimal refactoring solution with a number of iterations (stopping criteria) fixed to 2000 was less than forty-five minutes. This indicates that our approach is reasonably scalable from the performance standpoint, especially that quality improvements are not related in general to real-time applications where time-constraints are very important. Thus, we could conclude that our approach is scalable.

Even promising results are obtained; our proposal has some limitations that will address them in the future work. One of our approach inputs is a base of recorded/collected code changes on previous system versions. We believe that this data is sometimes not available, especially in the beginning of the projects. Instead, some recorded/collected changes can be used from other different systems. Thus, we are planning to extend our work to reuse refactoring collected from different projects. In addition, our results depend heavily on the quality of recorded/collected refactorings. We plan to add an additional step, before using the input data, to classify/filter the collected changes using different criteria (risky changes, requirements-based changes, etc.).

VI. Related Work

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.

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. [5] 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 behaviour (refactorings feasibility), but not the semantics domain. In addition, the validation was done only on the move method refactoring. O'Keeffe et al. [12] 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 [15], 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. In [1], Kessentini et al. have proposed a single-objective combinatorial optimization using genetic algorithm to find the best sequence of refactoring operations that improve the quality of the code by minimizing as much as possible the number of design defects detected on the source code.

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. In [2], the authors have proposed a multi-objective optimization approach to find the best sequence of refactorings using NSGA-II. This approach is based on two fitness functions: quality and effort. The quality corresponds to the number of corrected defects that are detected on the initial program, and the effort fitness function corresponds on the code modifications score. This approach recommends a sequence of refactorings that provide the best tradeoff between quality and effort. Recently, Ó Cinnéide et al. [31] have proposed a search-based refactoring to conduct an empirical investigation to assess some structural metrics and to explore relationships between them. To this end, they have used a variety of search techniques (Pareto-optimal search, semi-random search) guided by a set of cohesion metrics. To conclude, the vast majority of existing search-based software engineering approaches focused only on the program structure improvements and a maximum of two objectives are used. 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 a program that require refactoring by [14]. Opdyke [13] 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 postcondition verification. All these conditions could be expressed in terms of rules.

Our approach takes as input a set of recorded/collected refactorings applied to the system in the past. The easiest way to capture the applied refactorings is to track their execution in the development environment directly or to use diff-based techniques. Refactoring tracking is realized by [28] in programming environments and by [30] in modeling environments. In contrast to refactoring tracking approaches, state-based refactoring detection mechanisms aim to reveal refactorings a posteriori on the base of the two successively modified versions of a software artifact. Several approaches are tailored to detect refactorings in program code. For instance, Dig et al. [27] propose an approach to detect applied refactorings in Java code. They first perform a fast syntactic analysis and, subsequently, a semantical analysis in which also operational aspects like method call graphs are considered. In [25], a very recent approach for detecting refactorings improving several open issues of previous approaches has been proposed. In particular, the REF-FINDER tool, we used to collect refactorings for the experiments reported in this paper, is capable of detecting complex refactorings which comprise a set of atomic refactorings by using logic-based rules executed by a logic programming engine.

VII. Conclusion

We presented a search-based approach to improve the automation of refactoring. We used a set of recorded/collected code changes, combined with structural and semantic information, to improve the precision and efficiency of new refactoring suggestions. We start by generating some solutions that represent a combination of refactoring operations to apply. A fitness function combines three objectives: maximizing design quality, semantic coherence and similairty with pervious refactorings applied in similar contexts. To find the best trade-off between these objectives a NSGA-II algorithm is used. Our study shows that our technique outperforms state-of-the-art techniques where a maximum of two objectives are used.

The proposed approach was tested on open-source systems, and the results are promising. However, one of the main limitations of our work is how to deal with systems that did not have a history of applied refactorings. To solve this issue part of future work is to use collected refactorings from different systems and calculates a similarity with not only the refactoring type but also the context (code fragments).

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 IEEE ICPC11, pp. 81-90, Kingston, Canada, 2011.

[2] A. Ouni, M. Kessentini, H. Sahraoui and M. Boukadoum, Maintainability Defects Detection and Correction: A Multi-Objective Approach. Journal of Autmated Software Engineering, 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] A, Ouni, M, Kessentini, H, Sahraoui and M, S, Hamdi, Search-based Refactoring: Towards Semantics Preservation, 28th IEEE International Conference on Software Maintenance (ICSM), Riva del Garda- Italy, pp. 347–356, September 2012.

[5] 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.

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

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

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

[9] 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.

[10] 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.

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

[12] M. O'Keeffe, and M. O. Cinnéide, Search-based Refactoring for Software Maintenance. Journal of Systems and Software, pp.502–516.

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

[14] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin, Automated support for program refactorin using invariants, in Proc. Int'l Conf. Software Maintenance. 2001, pp. 736–743, IEEE Computer Society.

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

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

[17] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan, Optimizing Java bytecode using the Soot framework: Is it feasible? in International Conference on Compiler Construction (CC), 2000, pp. 18–34.

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

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

[20] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, JDeodorant: identification and application of extract class refactorings. International Conference on Software Engineering (ICSE), pp. 1037-1039, 2011.

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

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

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

[24] M. Kim, D. Notkin, D. Grossman, G. Jr. Wilson: Identifying and Summarizing Systematic Code Changes via Rule Inference, IEEE Transactions on Software Engineering, early access article, 2012.

[25] K. Prete, N. Rachatasumrit, N. Sudan, M. Kim: Template-based reconstruction of complex refactorings; in: Proceedings of the International Conference on Software Maintenance (ICSM'10), 2010.

[26] http://www-etud.iro.umontreal.ca/~ouniali/csmr2013/

[27] D. Dig, K. Manzoor, R. E. Johnson, T. N. Nguyen: Effective Software Merging in the Presence of Object-Oriented Refactorings, IEEE Transactions on Software Engineering 34 (3), pp. 321-335, 2008.

[28] T. Ekman, U. Asklund: Refactoring-aware Versioning in Eclipse. Electronic Notes in Theoretical Computer Science207, 57-69, 2004.

[29] R. Robbes: Mining a Change-Based Software Repository; in: Proceedings of the Workshop on Mining Software Repositories (MSR'07), IEEE Computer Society, pp. 15-23. 2008.

[30] M. Koegel, M. Herrmannsdoerfer, Y. Li, J. Helming, D. Joern: Comparing State- and Operation-based Change Tracking on Models; in: Proceedings of the IEEE International EDOC Conference, 2010.

[31] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. H. Moghadam, Experimental Assessment of Software Metrics Using Automated Refactoring, Proc. Empirical Software Engineering and Management (ESEM), pages 49-58, September 2012.

[32] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, and F. Palomba. Supporting Extract Class Refactoring in Eclipse: The ARIES Project. In Proceedings of the 34th International Conference on Software Engineering (ICSE), pp. 1419-1422, Zurich, Switzerland, 2012.

[33] A. Corazza, S D. Martino, and V. Maggio, LINSEN: An Efficient Approach to Split Identifiers and Expand Abbreviations, 28th IEEE International Conference of Software Maintenance (ICSM 2012), Riva del Garda (Trento),Italy, IEEE, pp. 233-242, 2012.