The Use of Development History in Software Refactoring Using a Multi-Objective Evolutionary Algorithm

Ali Ouni[1,2], Marouane Kessentini[2], Houari Sahraoui[1], 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]IT Department, Ahmed Ben Mohamed Military College, Qatar
mshamdi@yahoo.com

ABSTRACT
One of the widely used techniques for evolving software systems is refactoring, a maintenance activity that improves design structure while preserving the external behavior. Exploring past maintenance and development history can be an effective way of finding refactoring opportunities. Code elements which undergo changes in the past, at approximately the same time, bear a good probability for being semantically related. Moreover, these elements that experienced a huge number of refactoring in the past have a good chance for refactoring in the future. In addition, the development history can be used to propose new refactoring solutions in similar contexts. In this paper, we propose a multi-objective optimization-based approach to find the best sequence of refactorings that minimizes the number of bad-smells, and maximizes the use of development history and semantic coherence. 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 using different large open source projects.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Software – *Restructuring, reverse engineering, and reengineering*

General Terms
Measurement, Performance, Design, Experimentation

Keywords
Search-based Software Engineering, Refactoring, Semantics, Code Change, Design Defects

1. INTRODUCTION
Software undergoes continuous change, through which new features are added, bugs are fixed, and quality is improved during software development lifecycle. To support these activities, many tools emerged to manage source code such as concurrent versions system (CVS), and subversion (SVN) [23] where all documentation, configuration, and code-changes are archived and called "software development history".

GECCO'13, July 6–10, 2013, Amsterdam, The Netherlands.

Software-development history can be used to ease maintenance activities. One of the widely-used activities is *refactoring* which improves design structure without affecting its overall behavior. In general, to apply refactoring, we need to identify 1) *where* a program should be refactored and 2) *which* refactorings to apply [6] [22]. Automating the refactoring suggestion task is essential and very useful to help software developers in improving the quality of their code such as reusability, maintainability, flexibility, and understandability, etc.

Recently, search-based approaches have been applied to automate software refactoring [10] [5] [3] [4]. Most of these work formulated refactoring as a single-objective optimization problem, in which the main goal is to improve code quality while preserving the behavior (see for example, [1] [4] [10] [12]). In some other work, other objectives are also evaluated such as reducing the effort (number of code changes) [2], preserving semantic coherence [5], and improving quality metrics. However, sometimes structural and semantic information is not enough to generate efficient refactoring.

The use of development history can be an efficient solution to propose refactoring. Code fragments that are modified over the past in the same period are semantically connected (same feature). Furthermore, fragments that are extensively refactored in the past have a good probability for refactoring in the future. Moreover, the code to refactor can be similar to some patterns that can be found in the development history thus developers can easily adapt them. However, despite its importance, the history of code changes has not been explicitly investigated/used to suggest new refactorings. Existing approaches including recent work [2] [5] [1] suggests refactorings without considering software-development history.

In this paper, we propose a multi-objective optimization-based approach to find the best sequence of refactorings that minimizes the number of bad-smells, preserves the semantic coherence, and maximizes the consistence with development history. To this end, we use the non-dominated sorting genetic algorithm (NSGA-II) [8] to find the best trade-off between these three objectives. After generating refactoring solutions, we evaluate first if similar changes were applied in previous versions of the code fragments that will be refactored by the generated solution. Then, we check if a large number of changes were performed in the past on the code fragments to modify, and if the refactored code elements are, in general, modified at, approximately the same time or not (co-changed). In addition, we use a combination of quality metrics to detect bad-smells [2] 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) [5]. Thus, three objectives are considered in our NSGA-II adaptation: minimizing

the number of bad-smells, maximizing the semantic coherence and maximizing the use of development history information.

Our approach is evaluated on two large open source systems, and aimed at investigating to what extent can the use of code changes history improve the automation of code refactoring. We also evaluate the performance of our proposal by comparing it to existing contributions, random search and a single-objective algorithm without the use of the development history.

Thus, the paper presents the following research questions.

RQ1: To what extent the reuse of software development history can improve the results of refactoring suggestion compared to work that do not use them?

RQ2: How do the proposed multi-objective approach performs compared to random search, mono-objective approach and some existing work [1], [3], [2], and [15]?

The rest of this paper is structured as follows. Section 2 is dedicated to the background needed to understand our approach. In Section 3, we give an overview of our proposal and we describe how the historical code changes are used to guide the search-based refactoring. Then, in Section 4, 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 mining software version archives is outlined in Section 6. We conclude and suggest future research directions in Section 7.

2. BACKGROUND AND PROBLEM STATEMENT

To better understand our contribution, we start, in this section, by giving the definitions of important concepts. Then, we detail the specific problems that are addressed by our approach.

2.1 Background

Refactoring is a well-known technique frequently used during the whole software development cycle to improve software quality. Fowler [6] defines it as "A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior". One motivation is to reorganize classes, fields and methods in order to improve the software design and to facilitate future extensions. This reorganization is used to improve different aspects of software-quality: reusability, maintainability, complexity, etc. [22]. To apply refactoring, a software developer should identify where a program should be refactored and which refactorings to apply. In this paper, we identify refactoring solutions for the problem of correcting design defects that are detected in the source code. Hence, to detect design defects, we used detection rules, *i.e.*, combination of software metrics/thresholds, as an indicator for the presence of design defects [2]. We consider that these defects are already detected using defects detection rules [2], and we need to find what are the suitable refactoring operations to fix them.

Bad-smells, also called design defects, anomalies, design flaws, or anti-patterns [7] [6], refer to design situations that adversely affect the development of software, which mainly results of bad programming practices. As stated by Fenton and Pfleeger [7], bad-smells are unlikely to cause failures directly, but may do it indirectly. In general, they make a system difficult to maintain, to change and to evolve, which may, in turn, introduce bugs. They should be systematically found, and fixed all along the software lifecycle. The most well-known examples of bad-smells are the *blob* [6] which can be found in designs where one large class monopolizes the behavior of a large part of the system and other related classes primarily encapsulate data. For example, to correct the blob defect many refactorings can be used to reduce the number of functionalities in a specific class, such as moving methods and/or extracting a class.

2.2 Problem statement

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) [5]. 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). Thus, it is risky to only use 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.

Past maintenance and development history can be used to determine refactoring opportunities. Many aspects can help to improve the automation of refactoring: 1) code elements which undergo changes in the past, at approximately the same time, are in general semantically dependent, 2) code elements changed many times in the past have a good probability to be "badly-designed", 3) the development history can be used to propose new refactoring solutions in similar contexts. Recently, only the third aspect has been used by existing work [15].

2.3 Motivating scenario

To illustrate some of the above-mentioned issues, Figure 1 shows a concrete example extracted from Xerces-J [25], a well-known open-source family of software packages for parsing and manipulating XML.

We considered a design fragment that contains three classes *XIncludeHandler*, *ValidatorHandlerImpl*, and *XIncludeTextReader*. Using the detection rules [2], the class *XIncludeHandler* is detected as a blob (i.e., a large class that monopolizes the behavior of a large part of the system). One possible refactoring solution to improve the design quality is to move some methods and fields from this class to other classes in the program. In this way, we will reduce the number of functionalities implemented in the blob class *XIncludeHandler*. A refactoring is proposed to move the method *checkNotation*() from the defected class *XIncludeHandler* to another suitable class in a way that we preserve the semantic coherence of the original program. Based on semantic and structural information [5] many

other target classes are possible including *ValidatorHandlerImpl* and *XIncludeTextReader*. However, these two classes have approximately the same structure that can be formalized using quality metrics (e.g., number of methods, number of attributes, etc.). In addition, they are semantically close to *XIncludeHandler* using vocabulary-based measures, cohesion and coupling [5].

Based on the development change history found using CVS, the class *XIncludeHandler* has been never changed together with *ValidatorHandlerImpl* in any commit (code revision). However, we recorded that it has changed with the class *XIncludeTextReader* in more than 14 commits. As such, moving methods and/or attributes from the class *XIncludeHandler* to the class *XIncludeTextReader* has a higher correctness probability, and it is likely to be semantically coherent. Therefore, in this paper our hypothesis is that development change history is a valuable complement to semantic and structural information [5] to preserve the semantic coherence when applying refactoring.

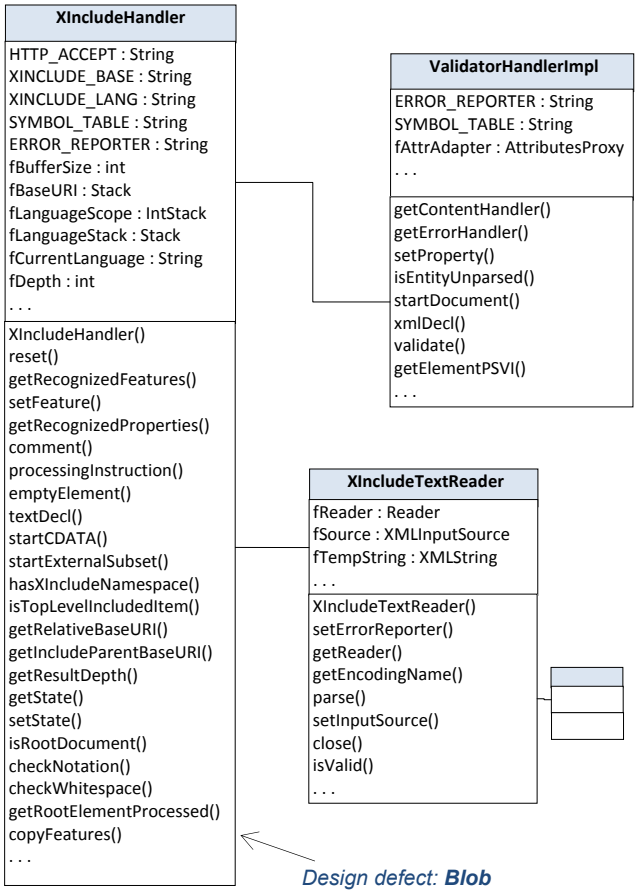


Figure 1. Design fragment extracted from Xerces-J v2.7.0

3. APPROACH OVERVIEW

The process of finding refactoring solutions, *i.e.*, sequence of refactoring operations, to correct bad-smells consist of exploring a huge search space. In fact, the search space is determined not only by the number of possible refactoring combinations, but also by the order in which they are applied. Thus, a heuristic-based optimization method is used to generate refactoring solutions. We have three objectives to optimize: 1) maximize quality improvement (bad-smells correction), 2) minimize the number of semantic errors, and 3) maximize the re-use with/of the development history. To this end, we consider the refactoring as a multi-objective optimization problem instead of a single-objective one. We propose an adaptation of the non-dominated sorting genetic algorithm (NSGA-II) [8]. This algorithm and its adaptation to the refactoring problem are described in section 4.

The search-based optimization process takes as inputs, a source code with defects, bad-smells detection rules [2], a set of possible refactoring operations, software version archive (*e.g.*, change log in CVS), recorded refactorings applied in the past, and a call graph for the whole program. As output, our approach recommends set of refactoring solutions. A solution consists of a sequence of refactoring operations that should be applied to correct the input code.

4. MULTI-OBJECTIVE SEARCH BASED REFACTORING

This section is dedicated to the description of our approach design. We describe how we encoded the optimization problem for the refactoring suggestion using NSGA-II.

4.1 NSGA-II overview

The basic idea of NSGA-II [8] is to make a population of candidate solutions evolve toward the near-optimal solution in order to solve a multi-objective optimization problem. NSGA-II is designed to find a set of optimal solutions, called non-dominated solutions, also Pareto set. A non-dominated solution is the one which provides a suitable compromise between all objectives without degrading any of them. As described in Algorithm 1, the first step in NSGA-II is to create randomly a 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 into an initial population R_0 of size N, and a subset of individuals is selected, based on the dominance principle and crowding distance [8] to create the next generation. This process will be repeated until reaching the last iteration according to stop criteria.

1. **while** *stopping criteria not reached* **do**
2. $R_t = P_t \cup Q_t$;
3. F = fast-non-dominated-sort (R_t);
4. $P_{t+1} = \emptyset$ and i=1;
5. **while** $| P_{t+1}| + |F_i| \leq N$ **do**
6. Apply crowding-distance-assignment(F_i);
7. $P_{t+1} = P_{t+1} \cup F_i$;
8. $i = i$+1;
9. **end**
10. $Sort(F_i, \prec n)$;
11. $P_{t+1} = P_{t+1} \cup F_i[1 : (N\text{-}| P_{t+1} |)]$;
12. Q_{t+1} = create-new-pop(P_{t+1});
13. $t = t$+1;
14. **end**

Algorithm 1: High-level pseudo-code of NSGA-II

4.2 NSGA-II adaptation

This section describes how NSGA-II [8] can be used for the refactoring suggestion problem. In general, to apply NSGA-II to a specific problem, the following elements have to be defined: representation of the individuals, evaluation of individuals using a fitness function for each objective to optimize, selection of the individuals to transmit from one generation to another, creation of

new individuals using genetic operators (crossover and mutation) to explore the search space, and generation of a new population.

4.2.1 Solution representation

To represent a candidate solution (individual), we used a vector representation. Each vector's dimension represents a refactoring operation. When created, the order of applying these refactorings corresponds to their positions in the vector. In addition, for each refactoring, a set of controlling parameters are randomly picked from the program to be refactored as illustrated in Table 1.

Ref	Refactorings	Controlling parameters
MM	Move Method	(source class, target class, method)
MF	Move Field	(source class, target class, field)
PUF	Pull Up Field	(source class, target class, field)
PUM	Pull Up Method	(source class, target class, method)
PDF	Push Down Field	(source class, target class, field)
PDM	Push Down Method	(source class, target class, method)
IC	Inline Class	(source class, target class)
IM	Inline Method	(source class, source method, target class, target method)
EM	Extract Method	(source class, source method, target class, extracted method)
EC	Extract Class	(source class, new class)

Table 1. Refactorings and its controlling parameters

An example of a solution is given in Figure 2. For short, we use abbreviations, e.g. MM for move method refactoring, EC for extract class (see Table 1). Of course, for each of these refactorings we specify pre- and post-conditions that are already studied in [11] to ensure the feasibility of applying these refactoring.

Figure 2. Representation of an NSGA-II individual

4.2.2 Fitness functions

After creating a solution, it should be evaluated using fitness function to ensure its ability to solve the problem under consideration. The solution proposed in this paper is based on studying how to preserve the way how code elements are semantically grouped and connected together when refactorings are decided automatically, and how to use the development change history to automate refactoring suggestion. Semantic preservation is captured by different heuristics/measures that could be integrated into existing refactoring approaches to help preserving semantic coherence.

Since we have three objectives, we define three different fitness functions:

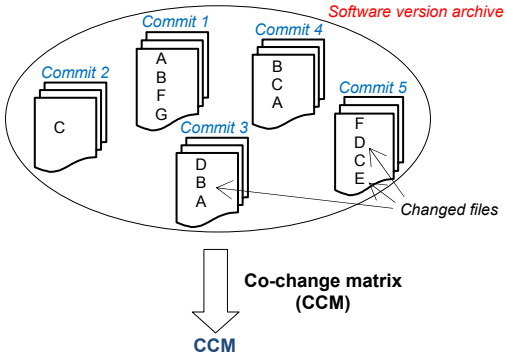
1) *Quality fitness function* that calculates the ratio of the number of corrected design defects (bad-smells) over the initial number of defects using detection rules [2].

2) S*emantic fitness function* that corresponds to the weighted sum of semantic measures [5]: vocabulary similarity (cosine similarity), and structural dependency (shared method calls and shared field access) used to approximate the semantic proximity between modified code elements. Hence, the semantic fitness function of a refactoring solution corresponds to the average of the semantic measure for each refactoring operation in the vector.

3) History of changes' fitness function that calculates an average of three measures: a) similarity with previous refactorings applied to similar code fragments, b) number of changes applied in the past to the same code elements to modify, and c) a score that characterizes the co-change of elements that will be refactored.

The first two fitness functions are studied before in [5] thus we are focusing in this paper on the third fitness function.

Recently, research has been carried out to mining software version repositories [20] [18] [16]. One of the important issues is to detect and interpret groups of software entities that change together. These co-change relationships have been used for different purposes: to identify hidden architectural dependencies [20], to point developers to possible places that need change [16], or to use them for software clustering [18]. Detecting historical co-change is mostly based on mining versioning systems such as CVS and in identifying pairs of changed entities. Entities are usually files and the change is determined through observing additions or deletions of lines of code. In the past decades, many CVS archives of open-source and industrial systems are freely available, e.g., via SourceForge.net.



	A	B	C	D	E	F	G
A	3	3	1	1	0	1	1
B	3	3	1	1	0	1	1
C	1	1	3	1	1	1	0
D	1	1	1	2	1	1	0
E	0	0	1	1	1	1	0
F	1	1	1	1	1	2	1
G	1	1	0	0	0	1	1

Figure 3. Co-changes Matrix

We analyze CVS archives to identify which classes have been changed together at the same time. To this end, we use only source code files to extract co-change; however, other non-useful files (e.g., documentation, configuration files, etc.) are filtered out. We consider, that when a source code file has been changed then each class in this file is changed (almost only one or two classes per file are implemented in actual systems). Each commit/revision contains a lot of information such as the file name, the commit/revision number, the developer who performed the commit, the log message, and the timestamp of the check-in. Co-change can be easily detected by mining version archives, and is less expensive than program analysis (coupling, cohesion, etc.) [21]. We depict in Figure 3 an example of how to detect the co-change between classes through different commits from version history archives to generate a co-change matrix CCM. $CCM_{i,j}$ represented how many times the class *i* have been changed at the same time with class *j* (co-change). For example, we can observe that almost when the class A change, the class B change also. This reveals that the implementations and semantics of these two classes are strongly related. Therefore, moving code elements (methods/attributes) between them is likely to be semantically

coherent since many changes have been manually performed between them in the past by software developers.

The second measure, *history_measure_2,* used in the history-based fitness function is the number of changes applied during the past to the same code elements to modify. If this number is high then we can conclude that this is a good indication that this code fragment is badly designed, thus represents a refactoring opportunity. This second measure is defined as:

$$history\ \_measure\ 2(RO_i) = \sum_{i=1}^{n} t(e) \quad (1)$$

where *t(e)* is the number of times that the code element(s) *e* was refactored in the past and *n* is the size of the list of possible refactoring operations.

The third measure of the history-based fitness function is defined to maximize/encourage the use of new refactorings 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:

$$history\ \_measure\ 3(RO) = \sum_{i=1}^{n} w * s \quad (2)$$

where *RO* is a suggested refactoring operation, *n* is the size of the list of recorded refactorings applied in the past, *s* is the number of times that a similar refactoring type has been applied in the past to the same code fragment/element and *w* is a refactoring weight (defined manually) that calculates the similarities between refactoring types if an exact matching cannot be found with the base of recorded refactoring.

4.2.3 Selection

To guide the selection process, NSGA-II sorts the population using the dominance principle and uses a comparison operator based on a calculation of the crowding distance [8] to select potential individuals to construct a new population P_{t+1}. Then, to generate an offspring population Q_{t+1}, 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 [26].

4.2.4 Crossover and Mutation

To better explore the search space, the crossover and mutation operators are defined. As illustrated in Figure 4, both operators are fairly simple. For crossover, we use a single-point crossover, and the mutation operator simply picks at random some positions in the vector and replaces them by other refactoring operations.

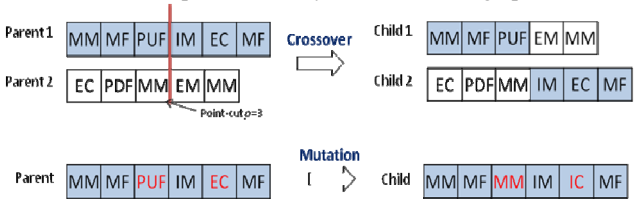


Figure 4. Crossover and mutation operators

5. VALIDATION

To evaluate the efficiency of our approach for generating good refactorings, we conducted experiments based on two large open-source systems to answer our two research questions described in section 1. We start by describing the designed our experiments. Then, we describe and discuss the obtained results.

5.1 Setup

We designed our experiments to address the above-mentioned two research questions. We apply our approach on two large open-source java projects JFreeChart [24] and Xerces-J [25]. JFreeChart is a powerful and flexible Java library for generating charts. Xerces-J is a family of software packages for parsing XML. Both programs and their change history are available through SourceForge.net. Table 2 provides some descriptive statistics about these two programs. To collect refactorings applied in previous versions, we use Ref-Finder [27], an Eclipse plug-in designed to detect refactorings between two program versions.

Systems	# classes	# defects	KLOC	# revision commits
Xerces v2.7.0	991	66	240	7493
JFreeChart v1.0.9	521	57	170	2009

Table 2. Program statistics

To answer **RQ1**, we validate manually the proposed refactoring operations to fix design defects. To this end, we calculate the defect correction ratio (DCR) given by Equation (3). It corresponds to the number of defects that are corrected after applying the suggested refactoring solution.

$$DCR = \frac{\#\,\text{corrected defects}}{\#\,\text{defects before applying refactorings}} \quad (3)$$

Then, we manually inspect the semantic correctness of the proposed refactoring operations for each studied system. We applied the proposed refactoring operations using ECLIPSE [28] and we check the semantic coherence of the modified code fragments. To this end, we define the metric refactoring precision (RP) that corresponds to the number of meaningful refactoring operations, in terms of semantic coherence, over the total number of suggested ones. RP is given by Equation (4).

$$RP = \frac{\#\,\text{coherent refactorings}}{\#\,\text{proposed refactorings}} \quad (4)$$

In our experiments, we calculate an average of the two-above mentioned scores over 30 runs to ensure the stability of our results.

In addition, to evaluate the efficiency of our approach, we compared our results to those produced by four existing contributions [3], [1], [2] and [15]. 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], a single-objective genetic algorithm is used to correct defects by finding the best refactoring sequence that reduces the number of defects. In [2], NSGA-II is used to find refactoring solutions to correct defects with low code modification/adaptation effort. And, in [15], a new objective is integrated to maximize the reuse of good refactorings applied to similar contexts. In all contributions, the development history is not used explicitly. We evaluate if our refactoring solutions produces more coherent changes than those proposed by [3], [1], [2] and [15] while having similar quality improvement.

To answer **RQ2**, we assessed 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).

Due to the stochastic nature of the algorithms/approaches we are studying, each time we execute an algorithm we can get slightly different results. To cater for this issue and to make inferential statistical claims, our experimental study is performed based on 31 independent simulation runs for each algorithm/technique studied. Wolcoxon rank sum test is applied between NSGA-II and each of the other algorithms/techniques (Kessentini et al., Ouni et al, MOGA, and Random Search) in terms of DCR with a 99% confidence level (α = 1%). Our tests shows that the obtained results are statistically significant with p-value<0.01 and not due to chance.

5.2 Results and Discussions

As described in Table 3, the majority of suggested refactorings by our approach improve significantly the code quality with acceptable correction scores compared to both other existing work. In addition, we preserve the semantics much better than all other approaches. For JFreeChart, 197 of the 210 proposed refactoring operations (94%) do not generate semantic incoherence. This score is higher than the one of the other approaches having respectively only 86%, 66%, 62% and 77% as RP scores. Thus, our multi-objective approach reduces the number of semantic incoherencies when applying refactoring operations. In the same time, after applying the proposed refactoring operations, we found that more than 86% (49/57) of detected defects were fixed. The corrected defects were of different types (blob, spaghetti code, and functional decomposition [2]). 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 other existing approaches that do not use development history (89%). Thus, the slight loss in the defect-correction ratio is largely compensated by the significant improvement of the semantic coherence.

Systems	Approach	Algorithm	DCR	RP
JFreeChart v1.10.2	***Our approach***	***NSGA-II***	***86% (49|57)***	***94% (197|210)***
	Ouni et al. '13 [15]	NSGA-II	82% (47|57)	86% (202|234)
	Harman et al. '07 [3]	Pareto opt.	N.A	66% (192|289)
	Kessentini et al. '11 [1]	GA	89% (51/57)	62% (147|236)
	Ouni et al. '12 [2]	NSGA-II	84% (48/57)	77% (157|203)
Xerces-J v2.7.0	***Our approach***	***NSGA-II***	***80% (53|66)***	***96% (282|294)***
	Ouni et al. '13 [15]	NSGA-II	79% (52|66)	93% (219|236)
	Harman et al. '07 [3]	Pareto opt.	N.A	63% (251|396)
	Kessentini et al. '11 [1]	GA	88% (58/66)	69% (212|304)
	Ouni et al. '12 [2]	NSGA-II	83% (55/66)	81% (186|228)

Table 3. Refactoring results

We also had similar results for Xerces-J. The majority of defects were corrected (80%), and most of the proposed refactoring sequences (96%) are coherent semantically. When comparing with previous work, Kessentini et al. performs slightly better for the number of corrected defects than NSGA-II (88%). However, for the refactoring precision, strategies proposed by our approach 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 our approach for Xerces-J are correct, whereas Kessentini et al., Harman et al. and Ouni et al. proposes respectively 69%, and 63%, 93% and 81% of good refactorings.

In conclusion, our approach produces good refactoring suggestions both from the point of views of defect-correction ratio and semantic coherence using the development change history.

In addition, we compare the refactoring results with and without use of development history. As illustrated in Figure 5, when only the quality fitness function is used, just 61% of proposed refactorings are semantically coherent for JFreeChart (only 69% for Xerces). Indeed, when the semantics is considered (vocabulary and structural information), the RP is improved, but it is still less than 81% for JFreeChart and 89% for Xerces. However, when the development history is considered, the majority of the proposed refactorings are semantically coherent, and we obtained 96% of RP for Xerces and 94% for JFreeChart.

Figure 5. Impact of the use of development history on refactoring suggestion

To answer RQ2, for evaluating the efficiency of NSGA-II and justifying 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 two fitness functions. In our mono-objective adaptation, we considered a single fitness function, which is the average score of the three objectives using genetic algorithm. Since in our NSGA-II adaptation we select a single solution without giving more importance to some objectives, we give equal weights for each fitness function value. As shown by Figure 6, NSGA-II outperforms significantly comparing to random-search and mono-objective algorithms in terms of corrected design defects and semantics preservation. For instance, in Gantt-Project NSGA-II has approximately the double of random/mono-objective RP and DCR scores.

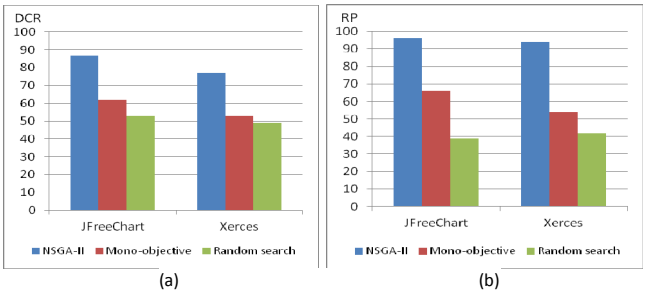


Figure 6. Comparison between random search, mono-objective, and multi-objective algorithms for refactoring suggestions: (a) Quality (DCR) and (b) Semantics preservation (RP).

Another element that should be considered when comparing the results of the three algorithms is that NSGA-II does not produce a single solution like GA, but a set of optimal solutions (non-

dominated solutions). The maintainer can choose a solution from them 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. In our case, the ideal solution has the best value of quality (equals to 1), of semantic coherence (equals to 1), and of change history reuse. Hence, we select the nearest solution to the ideal one in terms of Euclidian distance.

Finally, it is important to contrast the results of multiple executions. Over 31 independent simulation runs on JFreeChart, the average value of RP, DCR, development history reuse (collected refactorings) and execution time for finding the optimal refactoring solution with a number of iterations (stopping criteria) fixed to 8000 was respectively 94.86%, 86,83%, and 43min25s as shown in Figure 7. The standard deviation values was lower than 1, except for development history reuse (2.32). This indicates that our approach is reasonably scalable from the performance standpoint. Moreover, we evaluate the impact of the number of suggested refactorings on the RP, development history reuse, and DCR scores in 31 different executions. The best RP and DCR scores are also obtained with higher number of suggested refactorings. Thus, we could conclude that our approach is scalable from the performance standpoint, especially that quality improvements is not related in general to real-time applications where time-constraints are very important. In addition, the results accuracy is not affected by the number of suggested refactorings.

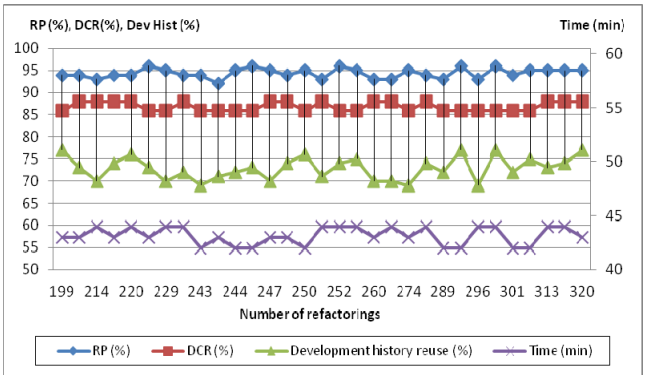


Figure 7. Impact of the number of refactorings on multiple executions on JFreeChart.

6. RELATED WORK

In this section, we review and discuss related work on the use of search-based refactoring, and the use of change history for several purposes in software engineering. We start by search-based refactoring that can be classified into two main categories: mono 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. [4] have proposed a single-objective optimization based on genetic algorithm (GA) to suggest a list of refactorings that maximizes a weighted sum of some quality metrics (coupling, cohesion, complexity, and stability). Similarly, O'Keeffe et al. [10] 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 [12], 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 approach is designed to find a good sequence of move method refactorings that should provide the best compromise between CBO and SDMPC to improve software quality. In addition, a multi-objective optimization approach [2] has been proposed 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 modification/adaptation score. This approach recommends a sequence of refactorings that provide the best tradeoff between quality and effort. Recently, Ouni et al. [5] have proposed a new multi-objective refactoring to find the best compromise between quality improvement and semantic coherence using two heuristics related to the vocabulary similarity and structural coupling. Later, in [15], the authors have integrated a new objective to maximize the reuse of good refactorings applied to similar contexts to propose new refactoring solutions. Ó Cinnéide et al. [14] 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.

Thus, the vast majority of existing search-based software engineering approaches focused only on the program structure improvements. However, the main limitation is that the semantics preservation is not considered in the search process. Moreover, suggesting new refactorings should not be made independently to previous changes and maintenance/development history. This is one of the most relevant limitations on search-based refactoring approaches, which do not consider how the software has been changed and evolved and how software elements are impacted by these changes.

In addition, data extraction from development history/repository is very well covered. Research has been carried out to detect and interpret groups of software entities that change together. These co-change relationships have been used for different purposes. Zimmermann et al. [16] have used historical changes to point developers to possible places that need change. In addition historical common code changes are used to cluster software artifacts [17] [18], to predict source code changes by mining change history [21] [16], to identify hidden architectural dependencies [20] or to use them as change predictors [19]. In addition, recently, co-change has been used in several empirical studies in software engineering. However, in the best of our knowledge, until now, the development change history is not used to suggest refactorings.

7. CONCLUSION

In this paper, we presented a novel search-based approach to improve the automation of refactoring. We used software-development history, 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 the re-use of the history of changes. 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 is 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.

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].

8. REFERENCES

[1] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni. Design Defects Detection and Correction by Example. In Proc. of the *19th IEEE Int. Conf. on Program Comprehension (ICPC)*, 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*, 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), 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), 1909-1916, 2006.

[5] A. Ouni, M. Kessentini, H. Sahraoui and M. S. Hamdi. Search-based Refactoring: Towards Semantics Preservation. In *Proceedings of the 28th IEEE International Conference on Software Maintenance* (ICSM), Italy, september 2012.

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

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

[10] M. O'Keeffe, and M. O. Cinnéide. Search-based Refactoring for Software Maintenance. *Journal of Systems and Software*, 81(4), 502–516, 2006.

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

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

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

[14] 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, 2012.

[15] A. Ouni, M. Kessentini and H. Sahraoui, Search-based Refactoring Using Recorded Code Changes, in *Proc of the 17th European Conference on Software Maintenance and Reengineering* (CSMR), Genova, Italy, march 5-8, 2013.

[16] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. 26th International Conference on Software Engineering* (ICSE), Edinburgh, Scotland, May 2004.

[17] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and R. Daniel. Using Concept Analysis to Detect Co-Change Patterns. In *Proceedings of International Workshop on Principles of Software Evolution* (IWPSE), 2007.

[18] D. Beyer, and A. Noack. Clustering Software Artifacts Based on Frequent Common Changes. *Proceedings of the 13th International Workshop on Program Comprehension*, 2005.

[19] A. Hassan and R. Holt. Predicting change propagation in software systems. In *Proceedings 20th Int. Conference on Software Maintenance* (ICSM'04), pp. 284–293, 2004.

[20] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. *Int. Proc. of Conf. on Soft. Maintenance* (ICSM), pages 190–198, Los Alamitos CA, 1998.

[21] T.T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Soft. Eng.* (TSE), Vol 30, No. 9, 2004.

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

[23] P. Cederqvist. Version Management with CVS, Dec. 2003. www.cvshome.org/docs/manual/

[24] http://www.jfree.org/jfreechart/

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

[26] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

[27] K. Prete, N. Rachatasumrit, N. Sudan, M. Kim. Template-based reconstruction of complex refactorings. in *Proc. of the Int. Conf. on Software Maintenance* (ICSM), 2010.

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