Explainable Search-Based Refactoring

Chaima Abid, Dhia Elhaq Rzig, Thiago Ferreira, Marouane Kessentini, and Tushar Sharma

Abstract—Refactoring is widely adopted nowadays in industry to restructure the code and meet high quality while preserving the external behavior. Many of the existing refactoring tools and research are based on search-based techniques to find relevant recommendations by finding trade-offs between different quality attributes. While these techniques show promising results on open-source and industry projects, they lack explanations of the recommended changes which can impact their trustworthiness when adopted in practice by developers. Furthermore, most of the adopted search-based techniques are based on random population generation and random change operators (e.g. crossover and mutation). However, it is critical to understand which good refactoring patterns may exist when applying change operators to either keep them or exchange with other solutions rather than destroying them with random changes. In this paper, we propose knowledge-informed change operators and an improved seeding mechanism that we integrated in a multi-objective genetic algorithm. We also provide explanations for refactoring solutions. First, we generate association rules using the Apriori algorithm to find relationships between applied refactorings in previous commits, their locations, and their rationale (quality improvements). Then, we use these rules to 1) initialize the population, 2) improve the change operators and seeding mechanisms of the multi-objective search in order to preserve and exchange good patterns in the refactoring solutions, and 3) explain how a sequence of refactorings collaborate in order to improve the quality of the system (e.g. fitness functions). The validation on large open-source systems shows that X-SBR provides refactoring solutions of a better quality than those given by the state-of-the-art techniques in terms of reducing the invalid refactorings, improving the quality, and increasing trustworthiness of the developers in the suggested refactorings via the provided explanations.

Index Terms—Refactoring recommendations, Search-Based Software Engineering, QMOOD metrics, multi-objective search

1 INTRODUCTION

As software systems continue to grow in size and complexity, their maintenance continues to become more challenging and costly [1], [2]. Several studies show that developers spend over 60% of their time in understanding existing code of large projects [3]. In order to improve the quality and maintainability of software systems, refactoring is widely adopted in industry to change the internal structure without affecting the external behavior of software systems [4].

A wide range of work has been done on finding refactoring recommendations using a variety of techniques including template/rule-based tools [5], [6], static and lexical analysis, and search-based software engineering [7]. Recent surveys show that search-based software engineering is widely adopted to find refactoring recommendations [7], [8] due to the conflicting nature of many quality metrics and the large search space of potential refactoring strategies that can be useful depending on the context. For instance, O'Keeffe et al. [9] compared the ability of different local search-based algorithms such as hill climbing and simulated annealing to generate refactoring recommendations that improve the QMOOD quality metrics [10]. Harman et al. proposed to use multi-objective search for refactoring to improve coupling and reduce cohesion [11]. Ouni et al. [12] and Mkaouer et al. [13] proposed multi-objective and many-objective techniques to balance different conflicting quality metrics when finding refactoring recommendations. Hall et al. [14] and Alizadeh et al. [15] improved the state-of-the-art of search-based refactoring by enabling interaction with the developers and learning their preferences. More detailed descriptions of existing search-based refactoring studies can be found in the following surveys [7], [8].

Despite the promising results of search-based refactoring on both open-source and industry projects, several limitations can still be addressed in order to improve their efficiency. These limitations can apply, in general, to most of the existing search-based software engineering studies [16]–[18] but we focus only on search-based refactoring in this paper. First, the random generation of the initial population can have a significant impact on the execution time and the quality of final solutions [19], [20]. Despite the large amount of data of the history of commits about applied refactorings, existing search-based refactoring studies are still generating the initial population of solutions randomly without exploiting the prior knowledge of what could construct a good refactoring solution. Second, most of software engineering problems, including refactoring, are discrete. However, the majority of existing studies are using regular change operators such as the random one-point crossover. The random application of change operators without understanding the good/bad patterns in a refactoring sequence of the solution can simply destroy them, deteriorate the quality, and delay the convergence towards good solutions. Third, current search-based refactoring techniques generate a large sequence of refactorings as one solution without explaining to developers how the different operations in the solution are dependent on each other in terms of fixing specific quality issues or improving the fitness functions which can impact their trustworthiness by developers in practice. Finally, the recommendation of refactorings is highly dependent to the

- *Chaima Abid, Dhia Elhaq Rzig, Thiago Ferreira, and Marouane Kessentini are with the department of Computer and Information Science, University of Michigan, Dearborn, MI, USA.*
E-mail: cabid@umich.edu, dhiarzig@umich.edu, thiagod@umich.edu, marouane@umich.edu
- *Tushar Sharma is with Siemens Corporate Technology, Charlotte, USA.*
E-mail: tusharsharma@ieee.org

Manuscript received December 19, 2020; revised December 19, 2020.

developers interest and preferences such as files owned or targeted quality goals. Thus, refactoring recommendations should be customized to the needs of the developers after understanding and learning their behavior and preferences.

In this paper, we propose an approach for refactoring recommendations based on a novel knowledge-informed multi-objective optimization algorithm to guide the generation of the initial population, define intelligent genetic operators and explain the generated refactoring solutions (also called the Pareto front). The proposed approach is a combination of an Apriori algorithm and multi-objective search. The first component of our approach is based on an Apriori algorithm [21] to generate association rules using the refactoring history and quality analysis of 18 projects of different sizes and categories. These association rules represent patterns linking a combination of refactoring types with their location, characterized using structural metrics, to their impact on improving the quality metrics/fitness functions (e.g. extendibility, functionality, flexibility etc.).

We evaluated our proposed algorithm on four open source projects and compared it with other variants of genetic algorithm with random initialization and/or genetic operators in terms of execution time of the algorithm, quality of the generated refactoring solutions and identified refactoring patterns. In addition, we performed a survey with 14 participants to check the correctness and relevance of the refactorings generated by the different algorithms using the four open source projects. Our focus in the human study was on selecting participants who are contributors/original developers of the used four open source systems. We found in our previous extensive studies on refactoring that the manual evaluation of non-original developers of the selected systems is not very useful as they are not knowledgeable enough about the projects to evaluate. We included only those four systems in our evaluation to attract the most amount of responses with good quality from participants. The more tedious the task that the participant must complete the less the quality of their input is. Furthermore, running all of the four algorithms on all of the systems 30 times is very time consuming. The results show that our technique performed significantly better than the four existing search-based refactoring approaches [9], [11], [22], [23] in terms of reducing the number of invalid refactorings, improving the quality, and increasing the level of trust between the developer and the refactoring tool. It also outperformed an existing refactoring tool that is not based on heuristic search, i.e., JDeodorant [24]. We used these four search-based refactoring techniques, the non-heuristic refactoring tool, and the four open source projects because 1) they are representative of existing automated multi-objective search-based refactoring techniques, 2) they are publicly available including the non search-based tool and 3) the familiarity of the participants with the open source systems that already part of an existing benchmark not constructed by the authors of this paper to avoid any potential bias [15]. We did not compare with manual and interactive refactoring techniques to ensure a fair comparison and focus on the scope of the contributions of this paper.

The remainder of this paper is organized as follows: Section 2 describes our enhanced knowledge-informed multi-objective search algorithm. Section 4 describes our validation methodology. Section 4 provides and discusses the different results obtained from our experiments. Section 5 presents the threats to validity. Section 6 discusses related work. Finally, Section 7 presents the conclusion and future works.

Replication Package. All material and data used in our study are available in our replication package [25].

2 X-SBR Approach

2.1 Overview

The goals of this paper are to 1) develop a knowledge-informed NSGA-II [26] by designing operators that prevent the destruction of good patterns in a solution 2) explain the decision made by the algorithm and give justifications to the users about why a refactoring solution can improve specific quality objectives by extracting the relevant patterns and 3) improve the population initialization by using the knowledge from the history of refactorings to create the individuals of the first generation rather than randomly generating them. To reach the stated goals, our approach takes as input the source code of several commits from different developers and projects and generates as output a Pareto front of refactoring solutions along with their explanations presented in a user friendly graphical interface. A refactoring solution is an ordered sequence of refactoring operations. The steps of our approach are as follows:

Step 1: Detect the refactoring history using Rminer [27] and compute quality metrics (described in section 2.2.1).

Step 2: Generation of association rules to link the quality metrics with refactoring operations collected in Step 1.

Step 3: Design of a knowledge-informed NSGA-II including the population generation and change operators based on the rules extracted in Step 2.

We note that only Step 3 needs to be executed on a new system to generate refactoring recommendations. Figure 1 summarizes our approach. It takes multiple commits of different systems that the developer worked on as input. For each commit, we analyze the source code automatically to extract low- and high-level quality metrics (refer to Table 2) and we extracted the refactoring using RMiner [27]. Based on the collected data, we applied the Apriori algorithm to find association rules to link low-level quality metrics and refactoring operations with high-level quality metrics. The rules are composed by two sides. The left-hand side includes only item-sets with elements belonging to the design properties (structure of the code) AND applied refactoring operations. The right-hand side needs to include only item-sets with elements belonging to the QMOOD metrics. The association rules were used to 1) initialize the first population of solutions, 2) select which refactorings of a solution to replace during crossover and mutation in order to avoid destroying good patterns and 3) explain the obtained refactoring sequence per solution to the developers by decomposing it to sub-sequences with their potential impact on quality improvements. Then, we designed and implemented a knowledge-informed NSGA-II to efficiently generate the initial population and perform change operators as detailed later. Finally, our approach can identify the specific refactoring patterns in each solution responsible

for the fitness values of each solution improvement or deterioration in the Pareto front.

2.2 Data collection

2.2.1 Quality Metrics

To evaluate the code quality of the systems, we selected the QMOOD model of Bansiya and Davis [10]. This hierarchical model defines six high-level quality metrics (described in Table 2) that are computed using a set of eleven weighted low-level object-oriented design metrics (described in Table 1). We selected this model as it has been extensively used in existing refactoring studies [9], [10], [15], [22], [28]–[30] as well as industrial projects [23], [31]–[37] to assess the quality of software systems. Thus, the paper is not making any new assumptions/validations about QMOOD.

TABLE 1: Design Metrics

Design Metric	Design Property	Description
Design Size in Classes (DSC)	Design Size	Total number of classes in the design.
Number Of Hierarchies (NOH)	Hierarchies	Total number of "root" classes in the design (*count(MaxInheritenceTree (class)=0)*)
Average Number of Ancestors (ANA)	Abstraction	Average number of classes in the inheritance tree for each class.
Direct Access Metric (DAM)	Encapsulation	Ratio of the number of private and protected attributes to the total number of attributes in a class.
Direct Class Coupling (DCC)	Coupling	Number of other classes a class relates to, either through a shared attribute or a parameter in a method.
Cohesion Among Methods of class ($CAMC$)	Cohesion	Measure of how related methods are in a class in terms of used parameters. It can also be computed by: *1 − LackOfCohesionOfMethods()*
Measure Of Aggregation (MOA)	Composition	Count of number of attributes whose type is user defined class(es).
Measure of Functional Abstraction (MFA)	Inheritance	Ratio of the number of inherited methods per the total number of methods within a class.
Number of Polymorphic Methods (NOP)	Polymorphism	Any method that can be used by a class and its descendants. Counts of the number of methods in a class excluding private, static and final ones.
Class Interface Size (CIS)	Messaging	Number of public methods in class.
Number of Methods (NOM)	Complexity	Number of methods declared in a class.

2.2.2 Extracting History of Refactorings

In this study, we used RMiner, a tool proposed by Tsantalis et al. [27], to extract the refactoring operations performed between Git commits. RMiner detects a total of 28 refactoring types at multiple granularity levels—Package, Type, Method, and Field. These types are the following: change package, extract and move method, extract class, extract interface, extract method, extract subclass, extract superclass, extract variable, inline method, inline variable, move and rename attribute, move and rename class, move attribute, move class, move method, move source folder, parameterize variable, pull up attribute, pull up method, push down attribute, push down method, rename attribute, rename class, rename method, rename parameter, rename variable, replace attribute, and replace variable with attribute.

We selected RMiner because it outperforms its competitors according to a survey of refactoring detection tools [38]. RMiner achieved accurate results in detecting refactorings compared to the state of-the-art tools, with a precision of 98% and recall of 87% [27]. RMiner was run on each project's source code as pulled from Github, the results generated were stored in CSV files. One line in the RMiner Data contained the Previous CommitID, Current CommitId, the Refactoring operation, Source class and Change in Quality Metrics after the application of the Refactoring. We provide in the validation section the details of the collected data related to refactorings and quality metrics on open source projects.

2.3 Association Rule Mining

Apriori is an algorithm for frequent item-set mining and association rule learning that was first defined by Agrawal et al. [21]. A frequent item-set is a set of items appearing together in a database meeting a user-specified threshold. The algorithm starts by finding the frequent individual items in a database and expand them to larger and larger item-sets as long as the appearance of those item-sets is larger than the threshold set by the user. The frequent item-sets found by Apriori can be used to generate association rules which highlight general trends in the database. The pseudo code of the Apriori algorithm can be found in the online appendix [25]. The Apriori algorithm takes as an input a series of transactions with discrete values of the different variables, thus a discretization process is necessary. We first scale the design and QMOOD variables with the Robust Scale method offered by Pandas [1]. Then, these variables were into discrete intervals using a combination of strategies: equal interval width, equal frequency, and k-means clustering. We kept the discretization results that generate the strongest rules in terms of confidence, support and lift.

In our study, the transaction database D consists of the list of classes of all commits that underwent a refactoring, their QMOOD/design metrics after discretization, and applied refactoring operations. The support threshold we considered was equal to 0.936. We say that a set of refactoring operations and QMOOD/design metrics is frequent if its support is 0.936 or more.

The number 0.936 was determined through trial and error: It is the lowest number that we used without getting an out of memory error. If we go any higher, we get too few rules, and lower we get an error. We defined three types of constraints on the generation of the rules:

- The left-hand side needs to include only item-sets with elements belonging to the design properties AND applied refactoring operations.
- The left-hand side needs to have at least 4 elements from the design properties item-set.
- The right-hand side needs to include only item-sets with elements belonging to the QMOOD metrics.

The goal of the first and second constraints is to include both the design metrics and the refactoring operations in the left-hand side of the rules to have a more relevant association of the refactoring operations with the high-level metrics. For example, we tend to apply the refactoring operator *Increase field Security* when the *Direct Access Metric*—ratio of the number of private and protected attributes to the

1. https://pandas.pydata.org/pandas-docs/stable/reference/index.html

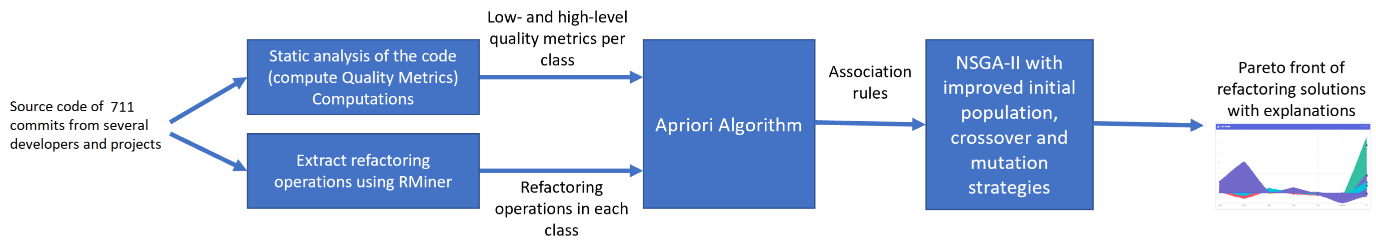


Fig. 1: Approach Overview

TABLE 2: QMOOD Metrics

Metric	Definition	Formula
Reusability	The ability of a design to be reused to a new problem without significant effort.	-0.25 × Coupling + 0.25 × Cohesion + 0.5 × Messaging + 0.5 × Design Size
Flexibility	The ability of a design to be adapted to provide functionality related capabilities easily.	0.25 × Encapsulation - 0.25 × Coupling + 0.5 × Composition + 0.5 × Polymorphism
Understandabili	The property of a design that enable it to be easily learned and comprehended.	-0.33 × Abstraction + 0.33 × Encapsulation + 0.33 × Coupling + 0.33 × Cohesion - 0.33 × Polymorphism - 0.33 × Complexity - 0.33 × Design Size
Functionality	The responsibility assigned to the classes of a design, which are made available by classes through their public interfaces.	0.12 × Cohesion + 0.22 × Polymorphism + 0.22 × Messaging + 0.22 × Design Size + 0.22 × Hierarchies
Extendibility	The ability of an existing design that allow for the incorporation of new requirements in the design easily.	0.5 × Abstraction - 0.5 × Coupling + 0.5 × Inheritance + 0.5 × Polymorphism
Effectiveness	This refers to the design's ability to achieve the desired functionality and behavior using object-oriented design concepts and techniques.	0.2 × Abstraction + 0.2 × Encapsulation + 0.2 × Composition + 0.2 × Inheritance + 0.2 × Polymorphism

Extract And Move Method, Inline Variable, CIS:(-2.484, 496.8], MOA:(-0.042, 8.4], NOH:(-0.0002, 0.0002], NOM:(-2.485, 497.0]→ Extendibility:(-0.2, 0.5], Flexibility:(-0.2, 0.5]

Fig. 2: Example of an association rule

total number of attributes in a class—is low. For the third constraint, since the fitness functions are composed only of QMOOD metrics, we made sure that the left hand side of the rules include only elements belonging to the QMOOD metrics. Figure 2 represents an example of one of the rules generated by the Apriori algorithm. The items in blue, red, and green are respectively the refactoring operations, design metrics, and QMOOD metrics, respectively. The rule can be interpreted as follows: when developers have applied the refactoring types *Extract and move method* and *Inline Variable* in a class that has the design metric CIS, MOA, NOH and NOM within the intervals of $(-2.484, 496.8]$, $(-0.042, 8.4]$, $(-0.0002, 0.0002]$, and $(-2.485, 497.0]$ respectively, then the change (as the difference between before and after refactoring) in extendibility and flexibility will be in the range of $(-0.2, 0.5]$, $(-0.2, 0.5]$ respectively. We designed a user-friendly interface in our web-app supporting the implementation of the approach proposed in this paper so the users can easily understand the explanations rather than reading mined association rules.

2.4 Knowledge-Informed and Explainable NSGA-II for Search-Based Refactoring

2.4.1 Proposed Algorithm

NSGA-II [26] is a well known, fast sorting multi-objective optimization algorithm that has been applied extensively to solve various optimization problems in software engineering [12], [13], [15], [39]. It tries to find non-dominated solutions, which cannot improve one objective without deteriorating others and exhibit different trade-offs between several conflicting objectives. In our study, the goal of the algorithm is to find non-dominated solutions balancing the six QMOOD quality metrics listed in Table 2. All the QMOOD metrics are to be maximized. The contributions of the paper are not about the use of QMOOD metrics for refactorings. The QMOOD metrics are used as the fitness functions of the multi-objective algorithm and the user/developer can change the fitness functions based on his/her preferences to evaluate the impact of refactorings. In this paper, our main contributions are the enhanced knowledge-informed seeding mechanism and change operators as well as the explainability of the obtained refactoring sequences to the developers. These contributions are independent from the used fitness functions and the algorithm can explain the obtained refactoring results based on whatever fitness functions that are used (QMOOD metrics or others).

We chose NSGA-II over NSGA-III based on the results of former studies [40]–[42]. We found that the performance of NSGA-II and NSGA-III are similar for the case of the refactoring problem using the 6 QMOOD objectives. NSGA-III generated a smaller number of non-dominated solutions than NSGA-II by around 18% (due to the nitching function of NSGA-III) but required the double execution time of NSGA-II. The manual evaluation with developers and automated evaluation using well-known metrics (IGD, Hyper Volume, quality improvements) showed that both algorithms generated similar results. This observation is explained by the fact that several QMOOD objectives are sharing the same low level quality metrics. Thus, the use of NSGA-II or NSGA-III will not make a difference for the particular problem addressed in this paper. The pseudo code of our adaptation of NSGA-II is presented in Algorithm 1. The search space consists of different refactoring operations applied to various code locations. Each operation is represented by an action (e.g., push down field, move method, move field, extract

sub class) and its parameters (e.g. source class, target class, attributes). A vector is used to represent a candidate solution. Each dimension represents a refactoring operation to apply. To assess the feasibility of solutions and see whether they maintain the behavior of the system. we used a set of pre- and post-conditions defined by Opdyke [43]. We did not define change operators that always generate feasible solutions because (1) a solution is a sequence of refactorings where the majority of them are dependent on each other thus it may not be possible to detect upfront those dependencies when generating solutions, (2) there is the possibility that the feasible space is fragmented and separated by infeasible regions, requiring that both the feasible and infeasible regions be searched. In fact, several authors have noted how the removal of infeasible solutions from the search space may cause diversity loss due to the additional selection pressure required to bias the population towards the feasible region [44]–[47]. The first step in NSGA-II is to create the initial population $P0$ of refactoring vectors. Then, a child population $C0$ is generated from the population of parents $P0$ 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 stopping criterion. To enable the static analysis of the source code, we used the Soot parser [48] which is a compiler framework for Java (bytecode). It allows the construction of a call graph which is a collection of edges representing all known method invocations in a system. It is noteworthy to mention that our approach can be applied to any object-oriented programming language. All that is needed is a parser that converts the code to a call graph. To calculate the fitness functions, the refactoring operations are applied automatically on the source code (to calculate the number of skipped invalid refactorings) then, we generate the call-graph. After that, we calculate the fitness functions by considering the changes in QMOOD values of the call graph before and after we apply the refactoring operations. We detail in the following three main components that we design to improve the regular NSGA-II algorithm: 1) the population generation; 2) change operators and 3) the explanations for the selected solution from the Pareto front.

2.4.2 Initial population

The initial population strategy is one of the important factors that affect the performance of search algorithms. The initial population has a key impact on the execution time and the quality of the generated Pareto front. Figure 3 summarizes the steps of the improved seeding mechanism. We first start by looking for all the rules, generated by the Apriori algorithm from the refactoring history, that can be applied to the classes of the system to be refactored. In other words, we look for the rules where there exist at least one class, from the system we are trying to refactor, with design metric values that satisfy/match the left-hand side of the rules. Then, we add all the refactoring operations of those rules in one unified pool. We note that we keep the refactorings of each rule as a group—also referred to as pattern—in a way that they are used together as a sub-sequence in the refactoring solution vector. The reason behind this grouping is that each group of refactorings tend to occur together according to the frequent

Algorithm 1 Pseudo code of Knowledge-Informed and Explainable NSGA-II adaptation for refactoring recommendation problem

```
1: Inputs: call graph of a software system S, refactoring operations TC
2: Output: subset(s) of the refactoring operations
3: Begin
4: I:= Instantiation(TC)// vectors of refactoring operations
5: P_0:=set_of(I)//Population Initialization
6: t:=0
7: Repeat
8: C_t:=apply_Genetic_Operators(P_t)//Apply the genetic operators on population P_t
9: for all I ∈ C_t do
10:    Extendibility(I):=calculate_Extendibility(I,S)
11:    Effectiveness(I):=calculate_Effectiveness(I,S)
12:    Functionality(I):=calculate_Functionality(I,S)
13:    Understandability(I):=calculate_Understandability(I,S)
14:    Flexibility(I):=calculate_Flexibility(I,S)
15:    Reusability(I):=calculate_Reusability(I,S)
16: end for
17: G_t:=P_t ∪ C_t // Combine parent and offspring populations
18: F:=fast_Non_Dominated_Sort(G_t) // F=(F_1,F_2,...), all nondominated fronts of G_t
19: P_{t+1} = ∅
20: i:=1
21: while |P_{t+1}| + |F_i| < Max_size do
22:    Crowding_distance_assignment(F_i) // calculate crowding distance in F_i
23:    P_{t+1}= P_{t+1} ∪ F_i // include i^th nondominated front in parent pop
24:    i:=i+1
25: end while
26: Sort (F_i, ≺_n) // sort in descending order using ≺_n
27: P_{t+1}= P_{t+1} ∪ F_i [1...(Max_size − |P_{t+1}|)] // choose the first Max_size - |P_{t+1}| elements of F_i
28: t:=t+1 // increment generation counter
29: until t=Max_iteration
30: best_solutions := first_front(P_t)
31: return best_solutions
```

item-set principle and the refactoring history of developers. Therefore, suggesting them together in a refactoring solution provides more personalized and practical recommendations. To create an initial population of size N, we randomly choose groups of refactorings from the pool we formed until we fill N ordered vectors.

2.4.3 Crossover

Figure 4 is a simplified illustration of how our improved crossover works. We first start by randomly picking two parents, S_1 and S_2, from the current population. S_1 and S_2 are vectors where each dimension represents a refactoring operation to apply. Then, we create cloning copies of the parents for the new pair of offspring S'_1 and S'_2. Next, we extract the Apriori rules that satisfy the following two conditions:

- The refactoring pattern in the left-hand side of the rule exists in S_1
- The design metric intervals in the left-hand side of the rule contain the values of the source class design metrics in the refactoring operations of S_1.

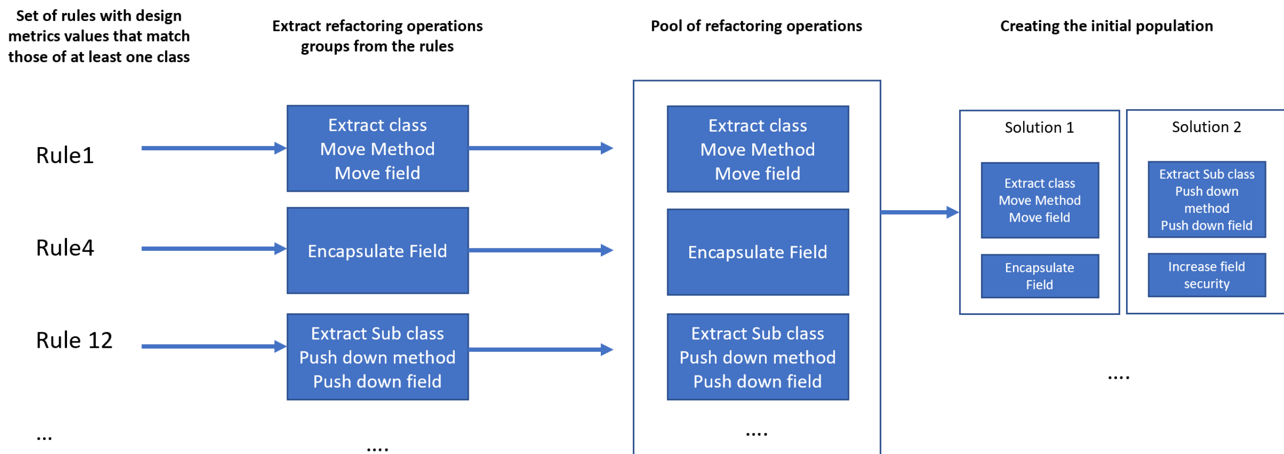


Fig. 3: Improved initial population process

We do the same for the second parent S_2. We end up having two rule sets R_1 and R_2 related to S_1 and S_2 respectively. Let O_1 and O_2 be the objectives (e.g. the QMOOD metrics) in the right-hand side of the rules in R_1 and R_2 respectively. Now, we compute the fitness function of S_1 and S_2 for all the objectives in $O_1 \cap O_2$ and we compare them. Let us consider that S_1 has a higher reusability than S_2. Thus, the algorithm will look for the rule R in R_1 that contains reusability in its right-hand side. We extract the refactoring operations from S_1 that match the refactoring pattern contained in R and transfer it to S'_2. We replace the genes of S_2 in S'_2 that are not used by any patterns contained in S'_2 for other objectives for which S_2 has a higher value in comparison to S_1. We do the same for all the objectives in $O_1 \cap O_2$. This crossover strategy allows us to keep the strengths and fix the weaknesses of the parents in the next generation while conserving the personalization aspect and practical abilities of the solutions.

2.4.4 Mutation

Mutation is a genetic operator used to preserve genetic diversity from one generation to the next in a genetic algorithm. Mutation involves a change in chromosome structure by altering one or more genes in a chromosome. It occurs according to a user-definable mutation probability. In our study, we set this probability to 0.1 . Figure 5 is a simplified illustration of how our improved mutation works. For each solution S, we randomly select a floating-point value. If this value is less than the mutation probability, we follow the steps below:

- We use the Apriori rules to find the refactoring patterns in S that improve one or more objectives. For example, in figure 5, Rule 2 improves Objective 2.
- We deduce the refactorings that are not associated with any pattern. In figure 5, the refactorings that are not associated with any objective are K, O, and L.
- We look for the rules that improve the weakest objective of S (i.e., the objective with the worst value). In figure 5, the weakest objective is Objective 1 which can be improved using Rule 1.
- We choose the refactoring pattern that modifies the maximum number of refactorings that are not associated with any objective and we add it to S. In figure 5, Rule 1 is composed of three refactorings that can replace the three refactorings that are not associated with any pattern (e.g. K, O, and L)
- If no rules are found, we choose a random number N between 1 and half the size of S and we randomly modify N refactorings in S from the possible refactoring operations that the tool supports.

2.4.5 Explanations Generation

Being able to explain and trust the outcome of a refactoring recommendation system is now a crucial aspect of the refactoring process and to ensure the trustworthiness of SBSE algorithms. In practice, developers tend to dismiss applying code changes if they do not understand why they need to be applied [49]. They may not want to take the time and effort to refactor a system without having a proper knowledge on the relationship between the quality metrics and the suggested refactorings [27]. In fact, refactoring is associated with costs such as testing the system after the changes are applied, thus developers will only apply the refactorings that they deem really important. To lift the lid of the black-box of the refactoring recommendation system, we provide explanations about how the solutions are formed. For each Pareto-optimal refactoring solution S, we look for the rules that satisfy the following two conditions:

- The refactoring pattern in the left-hand side of the rule exists in S
- The design metric intervals in the left-hand side of the rule contain the values of the source class design metrics in the refactoring operations of S.

The implemented tool includes several features to understand the explanations. First, the impact of the refactoring on the quality can be visualized to the developers via bar charts by showing the delta between before and after refactorings. Second, the extracted refactoring patterns are represented as dependencies tree to the developer and s/he can visualize

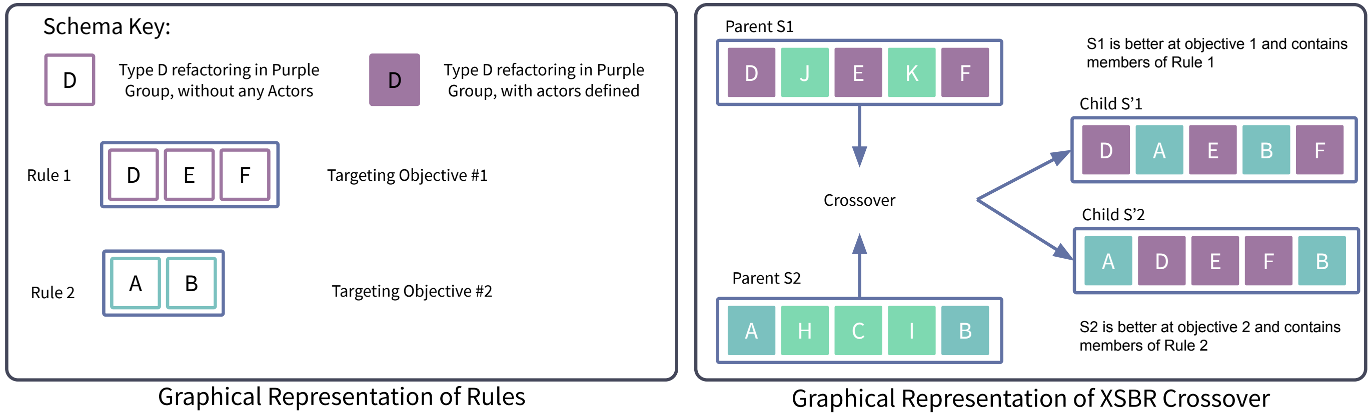


Fig. 4: An illustration of X-SBR crossover

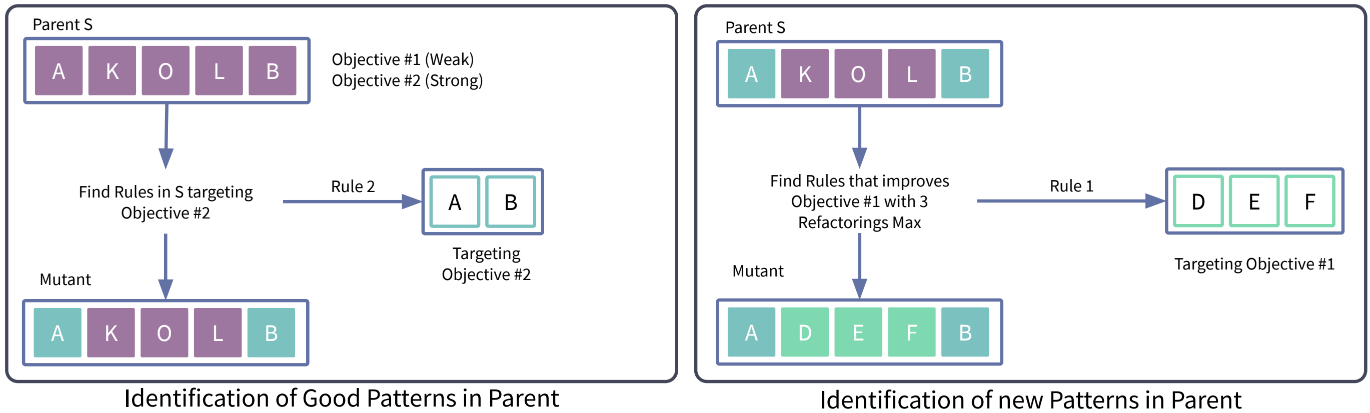


Fig. 5: An illustration of X-SBR mutation

the impact of each of the refactorings in the tree on the quality improvements or deteriorations. Third, the user can select any of the refactorings in the sequence and can get the pattern (other dependent refactorings with a significant impact on some of the quality metrics) associated with it to explain the relevance of that refactoring.

3 EXPERIMENTS

In this section, we first present our research questions and validation methodology. Then, we describe and discuss the obtained results. It should be emphasized that our technique is not dependent on specific objectives, algorithms and can be tested on other problem domains.

3.1 Research Questions

In this study, we defined three main research questions.

RQ1: *To what extent can X-SBR generate good refactoring solutions compared to multi-objective refactoring techniques in terms of performance indicators of genetic algorithms?*

RQ2: *To what extent can X-SBR reduce the number of invalid refactorings compared to multi-objective refactoring techniques?*

RQ3: *To what extent can X-SBR provide relevant solutions and explanations compared to the state of the art refactoring techniques?*

To answer RQ1, we collected the source code of 711 commits from 18 open-source systems. Table 3 contains the list of these open-source systems. We performed static analysis on the code to compute low- and high-level code quality metrics. Then, we used RMiner [27] to detect the refactoring operations performed between the commits. Our dataset can be found in the appendix website [25]. After that, we transformed the continuous design and QMOOD variables into discrete intervals. The High-Level metrics columns were divided into 6 bins. The rest of the columns were divided into 5 bins of flexible sizes that change in order to contain the highest number of values in each interval. Next, we used the Apriori algorithm [21] to generate association rules that link design metrics and refactoring operations with the QMOOD quality metrics. Then, we used these rules to choose strategically the initial population and improve the change operators of the traditional NSGA-II [22]. The rules are used to favor good patterns of the solutions and penalize bad ones.

To evaluate the efficiency of our algorithm, we selected four systems described in Table 4 since they are used in existing refactoring benchmark [22] and the participants of our study are familiar with them (RQ3). We compared four NSGA-II variations that optimize the same quality objectives: (1) traditional NSGA-II (Mkaouer et al. [22]) which is basically Algorithm 1, but with random initialization,

TABLE 3: List of projects used to extract the association rules

Project Name	GitHub Link	Number of commits	KLOC
Atomix	atomix/atomix.git	208	189.4
BTM	bitronix/btm.git	35	40.4
Erdos	Erdos-Graph-Framework/Erdos.git	1	7.6
FreeBuilder	inferred/FreeBuilder.git	30	7.6
Gson	google/gson.git	23	41.6
Hdiv	hdiv/hdiv.git	37	95
Hystrix	Netflix/Hystrix.git	42	85.4
Jolt	bazaarvoice/jolt.git	19	31.3
JSAT	EdwardRaff/JSAT.git	58	189.7
MinesJTK	MinesJTK/jtk.git	6	313.3
Pac4J	pac4j/pac4j.git	62	75.8
SecurityBuilder	tersesystems/securitybuilder.git	2	8.3
SMILE	haifengl/smile.git	25	2212.8
Tablesaw	jtablesaw/tablesaw.git	115	742.3
Tink	google/tink.git	15	356.3
OpenTracing Toolbox	zalando/opentracing-toolbox.git	8	15.7
JGraphT	jgrapht/jgrapht-capi.git	20	28.8
opencsv	jlawrie/opencsv.git	5	6.7

random mutation, and random crossover, (2) NSGA-II with an improved initial population strategy, random crossover and random mutation, (3) NSGA-II with improved change operators and random initial population strategy, and (4) NSGA-II with improved change operators and initial population strategy (*X-SBR*). To ensure a fair comparison, we only limited the baseline to these four techniques since our proposal is a variation of the work of Mkaouer et al. [22]. However, we extended our baseline in RQ3 when evaluating the relevance of the refactoring recommendations.

TABLE 4: Systems considered for validation

System	Release	# of Classes	KLOC
ArgoUML	v0.3	1358	114
JHotDraw	v7.5.1	585	25
GanttProject	v1.11.1	245	49
Apache Ant	v1.8.2	1191	112

To answer RQ2, we computed the number of conflicts in the solutions generated by the four algorithms mentioned above (RQ1) on the four systems listed in Table 4. For that, we calculated the number of invalid refactorings in each solution of the Pareto fronts by checking the validity of pre-and post-conditions of each refactoring operation.

To answer RQ3, we present to developers those association rules that lead to the generation of each refactoring solution in the Pareto front and their frequencies. Since the association rules are hard to understand if they are presented as the explanation for the recommended refactorings, we implemented a user-friendly interface in our refactoring webapp that can highlight the code locations and metrics associated with the recommended refactorings. To validate the usefulness of our explanations, we conducted a survey with a group of 14 active programmers to identify and manually evaluate the relevance of the refactorings that they found using *X-SBR*. Since the manual validation is limited to 14 participants, we considered another evaluation which is based on the percentage of fixed code smells (*NF*) by the refactoring solution. The detection of code smells after applying a refactoring solution is performed using the detection rules of [50]. The detection of code smells is subjective and some developers prefer not to fix some smells because the code is stable or some of them are not important to fix. To this end, we considered another metric based on QMOOD that estimates the quality improvement of the system by comparing the quality before and after refactoring independently from the number of fixed design defects. Based on the two above metrics, we can evaluate the different approaches without the need of developers evaluation. The baseline to answer RQ3 includes the different existing multi-objective techniques [9], [11], [22], [23] and also a tool, called JDeodorant [24], not based on heuristic search. All the selected search-based refactoring techniques for the baseline of RQ2 are based on multi-objective search but using different fitness functions and solution representation which may confirm if good refactoring recommendations are actually due to our knowledge-based component and not to the design of the algorithm. The current version of JDeodorant is implemented as an Eclipse plug-in that identifies some types of design defects using quality metrics and then proposes a list of refactoring strategies to fix them. For the comparison with JDeodorant, we limited the comparison to the same refactoring types supported by both X-SBR and JDeodorant which are the following: Move Method, Extract Method, and Extract Class.

3.2 Evaluation Metrics

To address the three research questions described in the introduction section, we defined the following metrics and applied them on a data set, described in the next subsection. For RQ1, we generated association rules that link design metrics and refactoring operations with QMOOD metrics. To evaluate these rules, we computed support, confidence, and lift [51].

Support: Support reflects how frequently the item set appears in the dataset. In our problem, it is defined as the ratio of the classes that contain $D \cup R \cup Q$ to the total number of classes in the dataset where D is a set of design metrics intervals, R is a set of refactoring operations and Q is a set of QMOOD intervals.

$$support(D, R \Rightarrow Q) = P(D \cup R \cup Q) \tag{1}$$

where P($D \cup R \cup Q$) is the probability of cases containing D, R and Q all in the same transaction.

Confidence: Confidence reveals how often the rule has been considered to be correct. In our approach, confidence is defined as the ratio of the number of classes that contain $D \cup R \cup Q$ to the number of classes that contain $D \cup R$. It evaluates the strength of a rule. The higher the confidence the more likely it is for Q to be present in transactions that contain $D \cup R$.

$$\begin{aligned} confidence(D, R \Rightarrow Q) &= P(Q|D \cup R) \\ &= {}^{P(D \cup R \cup Q)}/_{P(D \cup R)} \end{aligned} \tag{2}$$

Lift: Lift is defined as the confidence of the rule divided by the expected level of confidence. A lift value higher than 1 means that there is a positive correlation between $D \cup R$ and Q. If the lift is smaller than 1, it means that $D \cup R$ is negatively correlated with Q. A lift value almost equal to 1 means that we cannot say anything about the correlation of $D \cup R$ and Q.

$$\begin{aligned} lift(D, R \Rightarrow Q) &= {}^{confidence(D,R \Rightarrow Q)}/_{P(Q)} \\ &= {}^{P(D \cup R \cup Q)}/_{P(D \cup R) * P(Q)} \end{aligned} \tag{3}$$

To evaluate the quality of solution sets obtained by all four algorithms mentioned above, we used the following three metrics as performance indicators:

- *Contributions* (I_C) [52]: It measures the proportion of solutions that lie on the reference front RS (i.e., best know approximation set, computed as the non-dominated elements of all known solutions) [53]. The higher this proportion the better is the quality of the solutions.
- *Hypervolume* (I_{HV}) [54]: It computes the volume covered by members of a non-dominated set of solutions in the objective space. A higher value of hypervolume is desirable, as it demonstrates better spread and convergence of solutions.
- *Inverted Generational Distance* (I_{GD}) [55]: It computes the average Euclidean distance in the objective space between each solution in the Pareto front and its closest point in the reference front RS. Small values are desirable.

For RQ2, we want to estimate the feasibility of the solutions generated by the four algorithms. For that, we compute the number of invalid refactorings in each solution of the Pareto fronts by inspecting the validity of pre-and post-conditions of each refactoring operation. These conditions are discussed by Opdyke et al. [43]. We checked the pre and post-conditions automatically by verifying that (certain parts of) the behavior of the software is preserved by the refactoring. We have carefully validated the pre- and post-conditions of the refactoring types as part of previous studies [9], [43]. The exhaustive list can be found in the online appendix [25].

For RQ3, the goal is to validate the refactoring solutions generated by *X-SBR* from both quantitative and qualitative perspectives and compare them with those generated with baseline. For the quantitative validation, we calculated precision and recall scores to compare between refactorings suggested by *X-SBR* and those expected based on the participants assessment. We also did the same using the tools of the baseline.

$$Precision = \frac{\text{X-SBR solutions} \cap \text{Expected Refactorings}}{\text{X-SBR solutions}} \tag{4}$$

$$Recall = \frac{\text{X-SBR solutions} \cap \text{Expected Refactorings}}{\text{Expected Refactorings}} \tag{5}$$

For the qualitative validation, we asked the participants to assign 0 or 1 to every refactoring of the solutions generated by both tools. A 0 means that the refactoring is not applicable and inconsistent with the source code; 1 means that the refactoring is meaningful and relevant. We computed manual correctness which is defined as the number of meaningful refactorings divided by the total number of recommended refactorings. 1

$$\text{Manual Correctness} = \frac{|\text{Meaningful Refactorings}|}{|\text{Recommended Refactorings}|} \tag{6}$$

We have also calculate the number of code smells fixed by the recommended refactorings. Formally, *NF* is defined as:

$$NF = \frac{\#fixed\ code\ smells}{\#code\ smells} \in [0,1] \tag{7}$$

The gain for each of the considered QMOOD quality metrics and the average total gain in quality after refactoring can be easily estimated as:

$$G = \frac{\sum_{i=1}^{6} G_{q_i}}{6} \ and \ G_{q_i} = q'_i - q_i \tag{8}$$

where q'_i and q_i represents the value of the QMOOD quality attribute i after and before refactoring, respectively.

We finally asked the participants to evaluate the rules that are intended to explain the creation of the Pareto front solutions. For that, we randomly picked between 2 and 5 refactoring solutions per system and their explanations. Then, we asked them to assign a grade on a Likert scale of 1-5, 1 being the lowest (not relevant), 5 being the highest (very relevant) to every rule to indicate how helpful it is in explaining the creation and relevance of the refactoring solution.

3.3 Parameters tuning

Parameters setting plays an important role in the performance of a search-based algorithm. We have used one of the most efficient and popular approach for parameter setting of evolutionary algorithms which is Design of Experiments (DoE) [56]. Each parameter has been uniformly discretized in some intervals. Values from each interval have been tested for our application. Finally we pick the best values for all parameters. Hence, a reasonable set of parameter values have been experimented. We picked the combination based on the number of evaluations without improvement and convergence of the population. We tried to find a balance between wide exploration and deep exploitation during the evolutionary process. In order to ensure a fair comparison of the results of the four algorithms, we performed the same number of evaluations per run and used the same sizes for the initial population. We ended up by choosing 100 for the initial population and 10 000 for the maximum number of evaluations (the stopping criterion). We did not chose the execution time as a stopping criterion because it is known in the computational intelligence field that execution time is not suitable to ensure a fair comparison as it is very sensitive to the used hardware resources. The crossover and mutation probabilities are set to 0.8 and 0.1 respectively.

Because of the stochastic nature of the used meta-heuristic algorithms, different runs of the same algorithm solving the same problem typically lead to different results. For this reason, we performed 30 runs for each algorithm and each project to make sure that the results are statistically significant. For each evaluation metric, we used the Wilcoxon rank sum test [57] in a pairwise fashion in order to detect significant performance differences between the algorithms (X-SBR vs each of the competitors) under comparison based on 30 independent runs as recommended by existing guidelines [58].

We found that all the results based on the different measures were statistically significant on 30 independent runs using the Wilcoxon test with a 95% confidence level

($\alpha < 5\%$). The p-values of the pairwise analysis were lower than 0.01 in all cases. We have also calculated Eta squared (η^2) [59] which is a measure of the effect size (strength of association) and it estimates the degree of association between the independent factor and dependent variable for the sample. Eta squared is the proportion of the total variance that is attributed to a factor (the "refactoring methods" in this study). Table 5 reports Eta squared values for each pair of software projects and metrics. These values shows to what extent different algorithms are the cause of variability of the metrics.

TABLE 5: Effect Size values (Eta squared (η^2)) for corresponding software project and metric.

System	G	NF	MC	PR	RC
ApacheAnt	0.898	0.919	0.924	0.936	0.924
GanttProject	0.873	0.902	0.946	0.931	0.962
JHotDraw	0.826	0.903	0.918	0.836	0.962
ArgoUML	0.813	0.842	0.931	0.901	0.951

3.4 Subjects

We selected 14 participants to evaluate *X-SBR* on the four systems described in Table 4 because we wanted to target the right users of the tools to get actionable results. From our previous experience, in-depth interviews with relatively small targeted developers yield deep, quality insights that are more useful than the ones extracted using an online survey. In our study, we targeted participants who are part of the original developers of the systems that we used in our evaluation so they can give us valuable input and relevant feedback since they are very knowledgeable about the code. We advertised our study with 38 developers of the open-source projects selected based on their number of commits including the history of pervious refactoring (extracted using RefMiner). Out of the 21 participants who responded, we selected 16 of them based on the number of years of experience (should be more than a minimum of 5 years), current positions (required a current active position in industry), their knowledge of the systems beyond the system who served as a main contributor and availability for the manual assessments of the refactoring. Table 6 shows an overview of the background of the selected 14 participants after eliminating 2 participants due to their very limited knowledge of the only one system where they served as a contributor. They are all considered experts in refactoring in the development teams they work based on their several years of experience and extensive involvement in code rewriting projects.

THe participants had to fill a pre-study survey that collects background information on them such as their programming experience, their role within their companies etc. We divided the participants into 4 groups (2 groups of 3 and 2 groups of 4). The groups were formed based on the pre-study questionnaire and their familiarity with the studied systems to ensure that all the groups have almost the same average skill level. The details of the selected participants and the projects they evaluated can be found in Table 6 (the depicted values averages across the four participants in each row). To improve the survey outcome, we have made every possible effort to avoid any potential bias. We organized a two-hour lecture about software quality assessment in general and refactoring in particular. We also presented a demo for all the tools and gave them enough time to explore and test the tools themselves. We tested the trustfulness of participants and their knowledge on both the open source systems and refactoring beforehand by asking them to pass ten tests to evaluate their performance in evaluating and suggesting refactoring solutions. Each participant was asked to assess the meaningfulness and relevance of the refactorings recommended using our tool and all the four systems. The participants were shown recommendations created by the authors' approach as well as by the baseline, but without knowing which recommendations came from which approach. We assigned for each participant refactoring solutions from the different tools on the same system. Since the tools generate a lot of refactoring solutions, it is not possible to ask the participants to evaluate all of them. Therefore, to perform meaningful and fair comparisons, for each project and algorithm, we selected the solution using a knee-point strategy [60]. The knee point corresponds to the solution with the maximal trade-off between the different objectives which can be equivalent to the mono objective solution with equal objective weights if the objectives are not conflicting. Thus, we selected the knee point from the Pareto approximation having the median hyper-volume IHV value. The average number of refactorings per participant is 62. We ensured that each refactoring was evaluated by two developers and we considered it relevant if both of them agreed (The overall Cohen's kappa was 0.97). The experiment lasted between one to two hours.

TABLE 6: Participants details

System	#Subjects	Avg. prog. experience (years)	Refactoring experience
ArgoUML	4	10	High
JHotDraw	4	11.5	Very High
GanttProject	4	10.5	High
Apache Ant	4	12	Very High

4 Results

4.1 Results for RQ1

TABLE 7: Evaluation metrics and statistics of the rules

Evaluation Metric	Mean	Max	Min
Support	0.945	0.986	0.935
Confidence	0.986	0.992	0.959
Lift	1.000	1.002	0.999

We generated a total of 3097 association rules that link the design metrics and refactoring operations with the QMOOD quality metrics. Figure 2 shows an example of a rule created by the Apriori algorithm. The complete list can be found in our online appendix [25]. Table 7 contains the average, max and min support, confidence and lift of all the rules. The minimum support, confidence and lift are 0.935, 0.959 and 0.999, respectively. This confirms the strong correlation between design metrics, refactoring operations and the QMOOD metrics. After that, we compared the execution time of the four algorithms: (1) traditional NSGA-II (Mkaouer et al. [22]), (2) NSGA-II with an improved initial population

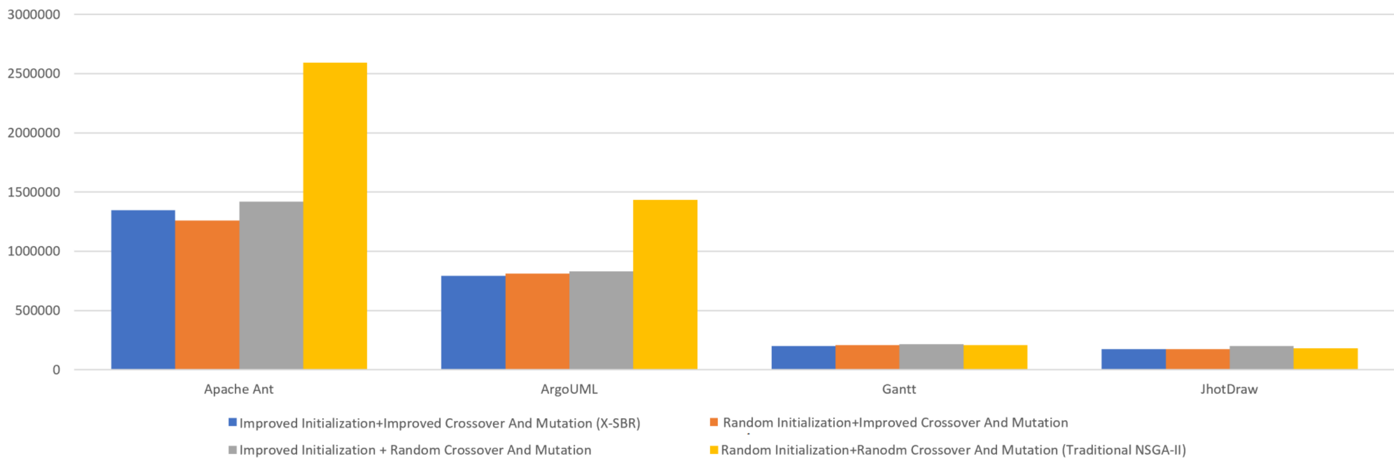


Fig. 6: Average execution time (ms) of all algorithms using the four systems

strategy, (3) NSGA-II with improved change operators, and (4) NSGA-II with improved change operators and initial population strategy (*X-SBR*).

Figure 6 shows the average time spent to run the four systems 30 times. In small systems (e.g. Gantt and JhotDraw), the four algorithms have almost the same execution time. X-SBR outperforms the other variations with a slight difference. However, when dealing with large systems (e.g. Apache Ant and ArgoUML), the traditional NSGA-II [22] has the highest execution time which is expected since both, the initialization and change operators, are done randomly without any guidance. This confirms the usefulness of our strategy of guiding the creation of solutions towards the construction of good refactoring patterns. The other three variations performed clearly better than the the traditional NSGA-II [22]. The difference in performance is more noticeable in large systems than in small systems because the execution time of the improved algorithms include the running time of the Apriori algorithm. Thus, the running time of the Apriori algorithm is compensated when we are dealing with a large number of classes by removing excessive diversity from the search space. Table 8 shows the mean, min and max of the Hypervolume (I_{HV}) and Generational Distance (I_{GD}) Indicators of all algorithms using the four systems. Table 9 contains the results of the Contribution (I_C) metric of the three modified algorithms compared to the traditional NSGA-II [22]. For each performance indicator, we highlighted in bold the best min/max/average values. Please note that the Contributions (I_C) and the Hypervolume (I_{HV}) are to be maximized and the Generational Distance (I_{GD}) is to be minimized. All these indicators show that the traditional NSGA-II exhibits more diversity in the solutions than other algorithms. This observation is expected as the traditional NSGA-II relies on randomness when generating the solutions, unlike the modified versions where the creation of solutions is guided towards the construction of good refactoring patterns based on the Apriori rules. It is important to note that excessive diversity can diverge the algorithm from generating good quality solutions due to the large search space and infinite number of possible combinations. In other words, we can end up having a diverse Pareto front but with many infeasible refactoring solutions. Therefore, it is necessary to have a strategy to push the algorithm towards creating correct solutions. However, guiding the algorithm too much might also hurt the exploration. Maintaining diversity is one important aim of multi-objective optimization. When clear user preferences are not available, it is highly desirable that a large number of solutions can be obtained that uniformly spread over the whole Pareto front and are as diverse as possible. However, we want to stay away from excessive diversity that leads the algorithm to diverge from generating good quality solutions due to the large search space and infinite number of possible combinations. On the other hand, selection pressure pushes the algorithm to focus more and more on the already discovered better performing regions in the search space and as a result population diversity declines, gradually reaching a homogeneous state. Through our approach, we are trying to maintain an optimal level of diversity in the population to ensure that progress of the search algorithm is unhindered by premature convergence to suboptimal solutions.

> ***Finding 1:*** The variants of NSGA-II with random initialization and/or genetic operators demonstrate higher diversity than *X-SBR* but the difference is small. *X-SBR* outperforms the other variations in terms of execution time, especially with large systems.

4.2 Results for RQ2

Figure 7 shows the average number of invalid refactorings in the solutions of the Pareto front in all four systems using the different algorithms. We would like to point out that all algorithms have the same number of non-dominated solutions in the final Pareto front which is equal to 100. The traditional NSGA-II and NSGA-II with random initialization and improved change operators had the largest number of invalid refactorings in their Pareto front with values exceeding 15 invalid refactorings. The lowest number of invalid refactorings was achieved by X-SBR. The latter algorithms had less than four invalid refactorings in their Pareto fronts. The reason why the combination of the random initialization and the random or improved crossover produce

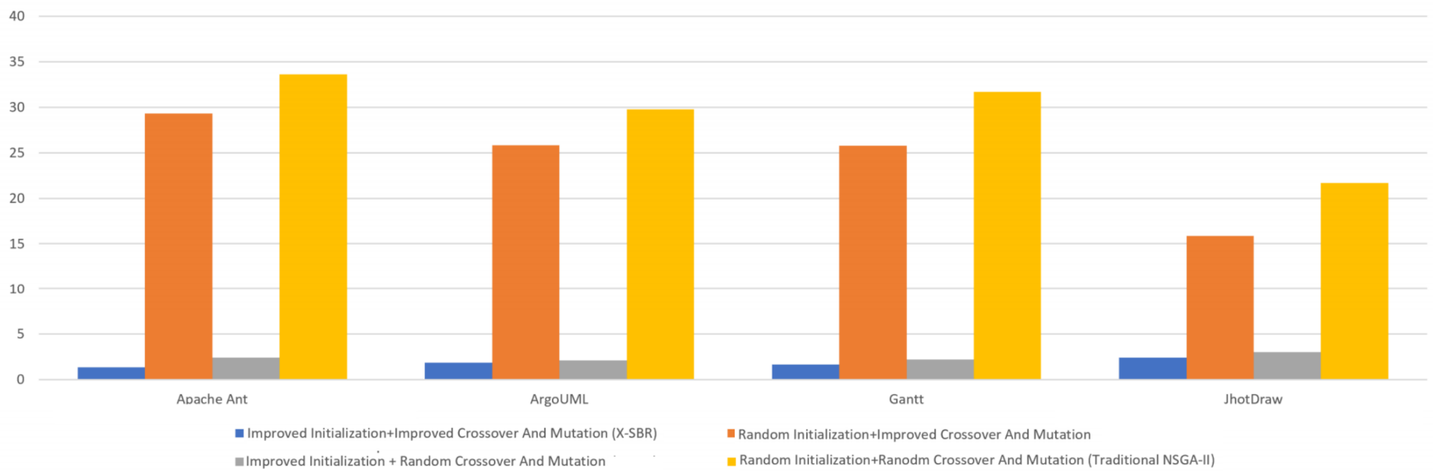


Fig. 7: Average number of invalid refactorings in the solutions of all algorithms using the four systems

a significant number of invalid refactorings is that the new crossover and mutation operators care more about improving the QMOOD quality metrics rather than checking the correctness of refactorings. However, this problem is mitigated by initializing the gene pool with valid chromosomes based on mining the refactoring history of several projects. This can be observed by the reduced number of infeasible refactorings in the solutions generated by the improved initialization method when combined with either the random or improved change operators.

> ***Finding 2:*** Based on the results of RQ1 and RQ2, *X-SBR* was able to achieve a better quality of solutions in comparison to the traditional NSGA-II with small sacrifices in terms of diversity and execution time.

4.3 Results for RQ3

We summarize in the following the feedback of the developers based on the survey. Figure 8 contains the results of the manual correctness, precision and recall of both our tool (*X-SBR*) and the state of the refactoring techniques. *X-SBR* was able to achieve better scores than [22] and existing approaches in all the previous metrics for all systems. The average manual correctness, precision and recall of our tool compared to that of Mkaouer et al. [22] are 0.839, 0.795, and 0.83 to 0.67, 0.56, and 0.67 respectively and much better than the remaining tools. The participants also found our refactoring recommendations applicable and consistent with the source codes and their design issues.

Figure 9 summarizes what the participants think about the explanations provided by *X-SBR*. For all the four systems, more than 85% of the rules are judged relevant (score 4) and very relevant (score 5). Only less than 3% of the rules were judged not relevant (score 1). They mentioned that *X-SBR* provided trust, clarity and understanding compared to existing refactoring tools. They highlighted that the black-box nature of existing refactoring tools, giving results without a reason, is hindering them from adopting their refactoring recommendations. According to them, this obstacle is alleviated by our proposed approach.

> ***Finding 3:*** *X-SBR* provided more relevant and meaningful refactorings than the state of the art refactoring techniques and helped the participants understand why and how the solutions are generated which boosted their trust in the refactoring tool.

5 Threats to Validity

Conclusion validity. The parameter tuning of the different search based algorithms used in our experiments creates an internal threat that needs to be evaluated in our future work. The parameters' values used in our experiments were found by trial-and-error [61]. It can be an interesting perspective to automate the parameter tuning process [58] for our approach so that the parameters are automatically set and updated during the execution in order to provide the best possible results and optimal performance.

Internal validity. The variation of correctness and speed between the different groups when using our approach and other tools is one potential internal threat. In fact, our approach may not be the only reason for the superior performance because the participants have different programming skills and familiarity with refactoring tools. To counteract this, we assigned the developers to different groups according to their programming experience so as to reduce the gap between the different groups and we also adopted a counter-balanced design. The relatively small number of participants might also be considered as a threat to the validity of our approach. We selected 14 developers to participate in our study because we wanted to target the right audience persona to get actionable results. Since the manual evaluation of the refactorings is subjective and depends on the programming style of developers, we considered quantitative metrics that are less subjective than the participants' opinion and can be automatically calculated with any bias of the human intervention. These metrics estimate

TABLE 8: Results of the Hypervolume (I_{HV}) and Generational Distance (I_{GD}) indicators

		Hypervolume (I_{HV})			Generational Distance (I_{GD})		
System	**Algorithm**	**Average**	**Min**	**Max**	**Average**	**Min**	**Max**
Apache Ant	Improved Initialization + Random Crossover And Mutation	0.680742	0.432318	0.935184	**0.015524**	0.010209	**0.020846**
Apache Ant	Random Initialization+Random Crossover And Mutation (Traditional NSGA-II)	0.693186	0.398396	1.117279	0.031465	0.00819	0.051153
Apache Ant	Improved Initialization+Improved Crossover And Mutation (*X-SBR*)	0.499433	0.293363	1.124596	0.064079	0.002978	0.093633
Apache Ant	Random Initialization+Improved Crossover And Mutation	0.809312	0.485615	1.085356	0.019873	0.008611	0.037001
ArgoUML	Improved Initialization + Random Crossover And Mutation	0.642199	0.404763	0.857439	0.024575	0.00818	0.034475
ArgoUML	Random Initialization+Random Crossover And Mutation (Traditional NSGA-II)	0.777583	0.52648	1.112845	0.03008	0.002679	0.047454
ArgoUML	Improved Initialization+Improved Crossover And Mutation (*X-SBR*)	0.641947	0.444483	1.136336	0.044481	0.002322	0.057297
ArgoUML	Random Initialization+Improved Crossover And Mutation	0.690078	0.444543	1.141642	0.041118	0.005496	0.055032
GanttProject	Improved Initialization + Random Crossover And Mutation	0.68693	0.566786	0.907115	0.021973	0.012951	0.029777
GanttProject	Random Initialization+Random Crossover And Mutation (Traditional NSGA-II)	0.861142	**0.666087**	1.133707	0.022668	0.002095	0.032585
GanttProject	Improved Initialization+Improved Crossover And Mutation (*X-SBR*)	0.782098	0.626555	0.978024	0.022406	0.011344	0.02651
GanttProject	Random Initialization+Improved Crossover And Mutation	0.776723	0.655532	1.242082	0.022349	0.006756	0.029976
JhotDraw	Improved Initialization + Random Crossover And Mutation	0.771315	0.588192	1.33879	0.04945	**0.000903**	0.071506
JhotDraw	Random Initialization+Random Crossover And Mutation (Traditional NSGA-II)	**0.933738**	0.555179	**1.281501**	0.026886	0.007105	0.056431
JhotDraw	Improved Initialization+Improved Crossover And Mutation (*X-SBR*)	0.564916	0.393056	1.083154	0.08246	0.024441	0.20624
JhotDraw	Random Initialization+Improved Crossover And Mutation	0.756657	0.592614	1.217705	0.052932	0.006605	0.072263

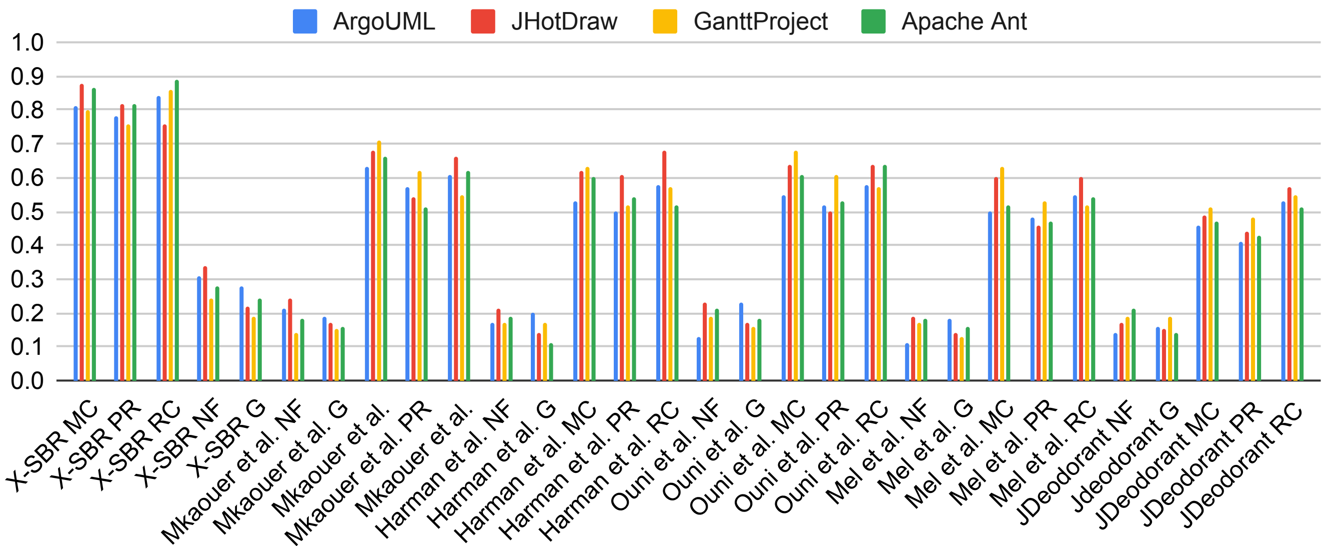


Fig. 8: Automated and manual evaluation of refactoring recommendations generated by the different refactoring tools

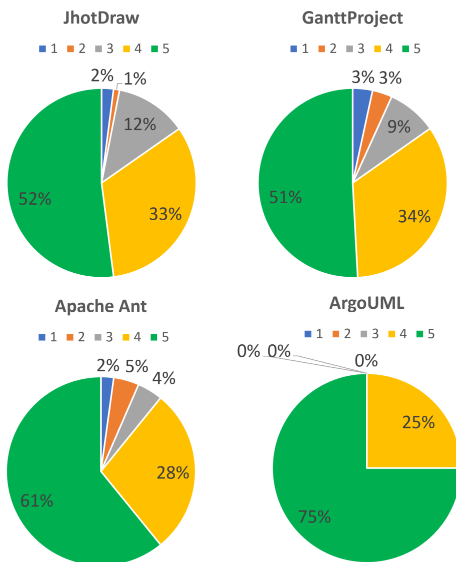


Fig. 9: Distribution of the relevance of the explanations according to the survey results (1=not relevant-5=very relevant)

TABLE 9: Results of the Contributions (I_C) metric

Algorithms	**Contribution value**
Contribution of NSGA-II with random initialization + improved change operators to traditional NSGA-II	**0.34030526**
Contribution of NSGA-II with improved initialization + random change operators to traditional NSGA-II	0.247601151
Contribution of NSGA-II with improved initialization + improved change operators to traditional NSGA-II	0.241613462

the quality improvement of the system by comparing the quality before and after applying the refactorings generated by the different approaches and also the number of fixed code smells. From our previous experience, in-depth interviews with a relatively small targeted audience persona yield deep, quality insights that are more useful than the ones extracted using an online survey. In our study, we targeted participants who are familiar with the systems that we used in our evaluation so they can give us valuable input and relevant feedback. We have also taken precautions to ensure that our participants represent a diverse set of software developers with experience in refactoring, and also that the groups

formed had, in some sense, a similar average skill set in the refactoring area.

Construct validity. The different developers involved in our experiments may have divergent opinions about the recommended refactorings in terms of relevance which may impact our results. Furthermore, we integrated our intelligent change operators in a multi-objective optimization algorithm, i.e, NSGA-II. It has been shown that NSGA-II demonstrates optimal performance when dealing with two or three objectives [62]. It has been also shown that increasing the amount of objectives impact the multi-objective evolutionary algorithms differently [62]. In our future work, we are planning to evaluate these change operators with other evolutionary algorithms as they should be algorithm independent.

External threats. External threats concern the generalization of our findings. We used 18 projects to generate the association rules. To mitigate these threats, we used projects of different sizes and domains. Moreover, we only included four projects in our validation. The reason behind that is, first, to attract the most amount of responses with good quality from participants in our survey. The more tedious the task that the participant must complete the less the quality of their input is. The second reason is the long execution time due to running all of the four algorithms on all of the systems 30 times. Furthermore, we considered 28 refactoring types, 11 design metrics, and 6 high-level quality metrics in our study. Finally, we only used the Apriori algorithm to link the design metrics and refactoring operations with the QMOOD metrics, instead of other features such as code smells.

6 RELATED WORK

In this section, we summarize the main research related to search-based software refactoring. We also highlight other search-based software engineering work that evaluated different strategies to seed the initial population and design the genetic operators. Table 10 outlines a summary of the differences between our approach and the existing studies.

6.1 Search-Based Software Refactoring

Many studies have used search-based techniques to automate software refactoring by optimizing different sets of quality metrics [9], [11], [17], [22], [23], [30], [69]–[78]. One interesting observation is that evolutionary algorithms are the dominant ones in search-based refactoring (e.g. NSGA-II, NSGA-III, etc.). Thus, we refer to evolutionary techniques when using the term search-based in this section. The reader can refer to the systematic literature review on search-based refactoring [7].

Kessentini et al. [50] proposed a single-objective combinatorial optimization using a genetic algorithm to find the best sequence of refactoring operations that improve the quality of the code by minimizing as much as possible the number of design defects detected in the source code. Harman and Tratt [11] were the first to use the concept of Pareto optimality in search-based software refactoring to deal with conflicting quality objectives such as coupling and cohesion. They showed that their multi-objective technique generates better results compared to a mono-objective approach. Ó Cinnéide et al. [30] proposed as well multi-objective search-based refactoring to conduct an empirical investigation to explore relationships between several structural metrics. They used different search techniques such as Pareto-optimal search and semi-random search guided by a set of cohesion metrics. Ouni et al. [63] presented a multi-objective refactoring formulation that generates solutions that maximize the number of detected defects after applying the proposed refactoring sequence and minimize the semantics similarity of the elements to be changed by the refactoring.

Several other studies aimed at addressing the refactoring problem while taking into account the developers' feedback and preferences in order to personalize the recommendations. Kessentini et al. [64] proposed a personalized multi-objective approach based on the analysis of the history of changes to recommend refactorings solutions that maximize refactoring operations for recently modified classes, classes containing incomplete refactorings detected in previous releases, and classes in bug reports. Wang et al. [65] proposed an interactive recommendation approach based on NSGA-II that suggests refactoring solutions for the interface of web services while taking the user feedback into consideration. These solutions optimize several interface design quality metrics such as coupling, cohesion, number of port types. They also fix antipatterns, maximize the interaction constraints learnt from the user feedback during the execution of the algorithm and minimize the deviation from the initial design. Alizadeh et al. [15] proposed an interactive and dynamic search-based approach to find refactoring solutions that improve software quality while minimizing the deviation from the initial design. The refactorings are ranked and suggested to the developer in an interactive fashion. The developer is allowed to accept, modify or reject any of the recommended refactorings. The feedback is then used to update the rankings of the refactoring solutions.

All the above studies used the traditional random change operators (e.g. 1-point crossover, random mutation, etc.). These change operators can destroy relevant patterns inside good refactoring solutions when applied randomly on discrete problems. Furthermore, the existing search-based refactoring studies are generating the initial population randomly, which may have a negative impact on the execution time and the quality of the final solutions. With the large amount of data on GitHub projects about refactorings applied by developers and their impact, it may be possible to inject good patterns extracted from the history of refactorings when generating the initial population or designing knowledge-based change operators which are the hypotheses of this paper.

It is possible that the injection of knowledge and preferences when generating the initial population can lead to less variety in the generated refactoring solutions, thus to less exploration of the search space. This is why we still used in our approach a random generation for part of the population to preserve variety in the initial population.

6.2 Seeding and genetic operators strategies in Search-Based Software engineering

Search-based software engineering (SBSE) studies proposed approaches on improving the initialization of the population

TABLE 10: Summary of related work

Paper	Software refactoring	Knowledge-based change operators	Personalization	Explainability
Kessentini et al. [50]	Yes	No	No	No
Ouni et al. [63]	Yes	No	No	No
Harman and Tratt [11]	Yes	No	No	No
Ó Cinnéide et al. [30]	Yes	No	No	No
Ouni et al. [12]	Yes	No	Yes	No
Ouni et al. [23]	Yes	No	Yes	No
Kessentini et al. [64]	Yes	No	Yes	No
Mkaouer et al. [13]	Yes	No	Yes	No
Wang at al. [65]	Yes	No	Yes	No
Alizadeh et al. [39]	Yes	Yes	Yes	No
Alizadeh et al. [15]	Yes	Yes	Yes	No
Fraser et al. [66]	No	Yes	No	No
Arcuri et al. [67]	No	Yes	No	No
Struber et al. [68]	No	Yes	No	No

and the design of change operators in order to optimize the performance and convergence of search algorithms as well as the quality of the generated solutions.

Fraser et al. [66] evaluated different strategies to seed the initial population in search-based techniques as well as techniques to seed values introduced during the search when generating tests for object-oriented code. They focused on three contexts: the first one is seeding of constants extracted from source code or bytecode throughout the search (e.g., initial population, mutation operators). The second one is related to strategies intended to improve the diversity of the initial population and its suitability for the optimization target. The last context targets the reuse of previously generated or hand-crafted solutions to seed the initial population of the search.

Arcuri et al. [67] conducted an empirical study in parameter tuning in search-based software engineering. They focused on test data generation for object-oriented software using the EVOSUITE tool. The results show evidence that parameter tuning is indeed critical, and very sensitive to the case study they provided.

7 Conclusion

Existing refactoring tools lack adaptability and explainability towards the developers. As a result, developers seem to be more inclined to abandon them and make changes by hand. We propose in this paper, *X-SBR*, an enhanced knowledge-informed multi-objective search algorithm to provide personalized and relevant refactoring recommendations. *X-SBR* implements new initial population and change operators methods using the refactoring and quality history of 18 projects and provides explanations regarding why and how the solutions are formed and impacted the fitness functions. Based on our quantitative and qualitative validation using four open-source systems, our tool was able to achieve more relevant refactoring solutions than existing refactoring techniques with a small sacrifice in terms of diversity and execution time. The results of the survey conducted with 14 software developers provide strong evidence that our tool improves the quality of refactoring solutions and helps developers understand, appropriately trust, and effectively manage the refactoring process.

There are multiple ways within which this work can be expanded upon. First, we believe it's a natural step to validate our work with additional programming languages, developers, projects, and quality metrics in order to draw conclusions about the general applicability of our methodology. Second, we intend to try out other algorithms for frequent item-set mining, beyond the Apriori Algorithm, to extend our empirical validation. Third, we think that adding support for more quality metrics and other fine-grained refactoring operations, such as Decompose Conditional, Replace Conditional with Polymorphism, and Replace Type Code with State/Strategy can prove an interesting addition and extension of our work. Fourth, we are planning to validate the change operators with other evolutionary algorithms such as a many-objective variant of MOEA/D, Global WASF-GA, and/or RVEA. We clarified this in the conclusion section. Last but not least, using code smell history and bug reports in addition to or in place of Low Level metrics when generating association rules can be an interesting future research direction

References

[1] G. Huang, H. Mei, and Q.-x. Wang, "Towards software architecture at runtime," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 2, p. 8, 2003.

[2] S. Das, W. G. Lutters, and C. B. Seaman, "Understanding documentation value in software maintenance," in *Proceedings of the 2007 Symposium on Computer human interaction for the management of information technology*, 2007, pp. 2–es.

[3] C. A. C. Coello, G. B. Lamont, D. A. Van Veldhuizen *et al.*, *Evolutionary algorithms for solving multi-objective problems*. Springer, 2007, vol. 5.

[4] M. Fowler, *Refactoring: Improving the Design of Existing Programs*, 1st ed. Addison-Wesley Professional, 1999.

[5] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha, "Recommending refactorings to reverse software architecture erosion," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 335–340.

[6] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 268–278.

[7] T. Mariani and S. R. Vergilio, "A systematic review on search-based refactoring," *Information and Software Technology*, vol. 83, pp. 14–34, 2017.

[8] M. Mohan and D. Greer, "A survey of search-based refactoring for software maintenance," *Journal of Software Engineering Research and Development*, vol. 6, no. 1, p. 3, 2018.

[9] M. O'Keeffe and M. O. Cinnéide, "Search-based refactoring for software maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502–516, 2008.

[10] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[11] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 2007, pp. 1106–1113.

[12] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and M. S. Hamdi, "Improving multi-objective code-smells correction using development history," *Journal of Systems and Software*, vol. 105, pp. 18–39, 2015.

[13] M. W. Mkaouer, M. Kessentini, M. Ó. Cinnéide, S. Hayashi, and K. Deb, "A robust multi-objective approach to balance severity and importance of refactoring opportunities," *Empirical Software Engineering*, vol. 22, no. 2, pp. 894–927, 2017.

[14] M. Hall, N. Walkinshaw, and P. McMinn, "Supervised software modularisation," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 472–481.

[15] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai, "An interactive and dynamic search-based approach to software refactoring recommendations," *IEEE Transactions on Software Engineering*, 2018.

[16] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, 2009.

[17] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2009.

[18] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue, "Search-based software library recommendation using multi-objective optimization," *Information and Software Technology*, vol. 83, pp. 55–75, 2017.

[19] V. Toğan and A. T. Daloğlu, "An improved genetic algorithm with initial population strategy and self-adaptive member grouping," *Computers & Structures*, vol. 86, no. 11-12, pp. 1204–1218, 2008.

[20] Y. Deng, Y. Liu, and D. Zhou, "An improved genetic algorithm with initial population strategy for symmetric tsp," *Mathematical Problems in Engineering*, vol. 2015, 2015.

[21] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.

[22] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó. Cinnéide, and K. Deb, "On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2503–2545, 2016.

[23] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, pp. 1–53, 2016.

[24] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE, 2008, pp. 329–331.

[25] A. authors. (2020) Study appendix. URL: https://sites.google.com/view/tse2020xsbr.

[26] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[27] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 483–494.

[28] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*, 1st ed. Morgan Kaufmann, 2014.

[29] 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 8th annual conference on Genetic and evolutionary computation*, 2006, pp. 1909–1916.

[30] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. Hemati Moghadam, "Experimental assessment of software metrics using automated refactoring," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, 2012, pp. 49–58.

[31] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum, "Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes," *IEEE Transactions on software Engineering*, vol. 33, no. 6, pp. 402–419, 2007.

[32] S. S. Virani, S. Messimer, P. Roden, and L. Etzkorn, "Software quality management tool for engineering managers," in *Proceedings of the Industrial Engineering Research Conference*, 2008, pp. 1401–1406.

[33] G. Samarthyam, G. Suryanarayana, T. Sharma, and S. Gupta, "Midas: A design quality assessment method for industrial software," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 911–920.

[34] N. F. I. A. Hamid and M. K. Hasan, "Identifying software quality factors for telecommunication industry in malaysia," in *Proceedings of the 2011 International Conference on Electrical Engineering and Informatics*. IEEE, 2011, pp. 1–5.

[35] ——, "Industrial-based object-oriented software quality measurement system and its importance," in *2010 International Symposium on Information Technology*, vol. 3. IEEE, 2010, pp. 1332–1336.

[36] A. Almogahed *et al.*, "Empirical studies on software refactoring techniques in the industrial setting," *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, vol. 12, no. 3, pp. 1705–1716, 2021.

[37] S. Gupta, H. K. Singh, R. D. Venkatasubramanyam, and U. Uppili, "Scqam: a scalable structured code quality assessment method for industrial software," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 244–252.

[38] L. Tan and C. Bockisch, "A survey of refactoring detection tools." in *Software Engineering (Workshops)*, 2019, pp. 100–105.

[39] V. Alizadeh and M. Kessentini, "Reducing interactive refactoring effort via clustering-based multi-objective search," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 464–474.

[40] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using nsga-iii," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, 2014, pp. 1263–1270.

[41] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using nsga-iii," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 3, pp. 1–45, 2015.

[42] M. Fleck, J. Troya, M. Kessentini, M. Wimmer, and B. Alkhazi, "Model transformation modularization as a many-objective optimization problem," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1009–1032, 2017.

[43] W. F. Opdyke, "Refactoring object-oriented frameworks," 1992.

[44] S. B. Hamida and A. Petrowski, "The need for improving the exploration operators for constrained optimization problems," in *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No. 00TH8512)*, vol. 2. IEEE, 2000, pp. 1176–1183.

[45] A. H. Aguirre, S. B. Rionda, and C. A. C. Coello, "Passss: an implementation of a novel diversity strategy for handling constraints," in *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, vol. 1. IEEE, 2004, pp. 403–410.

[46] A. H. Aguirre, S. B. Rionda, C. A. Coello Coello, G. L. Lizárraga, and E. M. Montes, "Handling constraints using multiobjective optimization concepts," *International Journal for Numerical Methods in Engineering*, vol. 59, no. 15, pp. 1989–2017, 2004.

[47] E. Mezura Montes, "Alternative techniques to handle constraints in evolutionary optimization," 2004.

[48] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, vol. 15, 2011, p. 35.

[49] E. R. Murphy-Hill and A. P. Black, "Why don't people use refactoring tools?" in *WRT*, 2007, pp. 60–61.

[50] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, 2011, pp. 81–90.

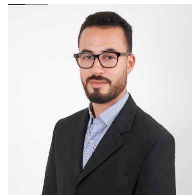
[51] P. D. McNicholas, T. B. Murphy, and M. O'Regan, "Standardising the lift of an association rule," *Computational Statistics & Data Analysis*, vol. 52, no. 10, pp. 4712–4721, 2008.

[52] F. Ferrucci, M. Harman, J. Ren, and F. Sarro, "Not going to take this anymore: multi-objective overtime planning for software engineering projects," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 462–471.

[53] H. Meunier, E.-G. Talbi, and P. Reininger, "A multiobjective genetic algorithm for radio network optimization," in *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No. 00TH8512)*, vol. 1. IEEE, 2000, pp. 317–324.

[54] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach," *IEEE transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.
[55] D. A. Van Veldhuizen and G. B. Lamont, "Multiobjective evolutionary algorithm research: A history and analysis," Citeseer, Tech. Rep., 1998.
[56] E.-G. Talbi, *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009, vol. 74.
[57] F. Wilcoxon, S. Katti, and R. A. Wilcox, "Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test," *Selected tables in mathematical statistics*, vol. 1, pp. 171–259, 1970.
[58] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 1–10.
[59] J. T. Richardson, "Eta squared and partial eta squared as measures of effect size in educational research," *Educational Research Review*, vol. 6, no. 2, pp. 135–147, 2011.
[60] X. Zhang, Y. Tian, and Y. Jin, "A knee point-driven evolutionary algorithm for many-objective optimization," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 6, pp. 761–776, 2014.
[61] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, pp. 1–61, 2012.
[62] S. Ong and A. Täcklind, "Performance differences between multi-objective evolutionary algorithms in different environments," 2016.
[63] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "Search-based refactoring: Towards semantics preservation," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 347–356.
[64] M. Kessentini, T. J. Dea, and A. Ouni, "A context-based refactoring recommendation approach using simulated annealing: two industrial case studies," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2017, pp. 1303–1310.
[65] H. Wang, M. Kessentini, and A. Ouni, "Interactive refactoring of web service interfaces using computational search," *IEEE Transactions on Services Computing*, 2017.
[66] G. Fraser and A. Arcuri, "The seed is strong: Seeding strategies in search-based software testing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 121–130.
[67] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *International Symposium on Search Based Software Engineering*. Springer, 2011, pp. 33–47.
[68] D. Strüber, "Generating efficient mutation operators for search-based model-driven engineering," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2017, pp. 121–137.
[69] M. Kessentini, H. Sahraoui, and M. Boukadoum, "Example-based model-transformation testing," *Automated Software Engineering*, vol. 18, no. 2, pp. 199–224, 2011.
[70] M. Kessentini, M. Wimmer, H. Sahraoui, and M. Boukadoum, "Generating transformation rules from examples for behavioral models," in *Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications*, 2010, pp. 1–7.
[71] A. Ghannem, M. Kessentini, and G. El Boussaidi, "Detecting model refactoring opportunities using heuristic search," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, 2011, pp. 175–187.
[72] S. Kalboussi, S. Bechikh, M. Kessentini, and L. B. Said, "Preference-based many-objective evolutionary testing generates harder test cases for autonomous agents," in *International Symposium on Search Based Software Engineering*. Springer, Berlin, Heidelberg, 2013, pp. 245–250.
[73] A. Ghannem, G. El Boussaidi, and M. Kessentini, "Model refactoring using examples: a search-based approach," *Journal of Software: Evolution and Process*, vol. 26, no. 7, pp. 692–713, 2014.
[74] M. Kessentini, A. Ouni, P. Langer, M. Wimmer, and S. Bechikh, "Search-based metamodel matching with structural and syntactic measures," *Journal of Systems and Software*, vol. 97, pp. 1–14, 2014.
[75] U. Mansoor, M. Kessentini, P. Langer, M. Wimmer, S. Bechikh, and K. Deb, "Momm: Multi-objective model merging," *Journal of Systems and Software*, vol. 103, pp. 423–439, 2015.
[76] R. Almhana, W. Mkaouer, M. Kessentini, and A. Ouni, "Recommending relevant classes for bug reports using multi-objective search," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 286–295.
[77] M. Kessentini and A. Ouni, "Detecting android smells using multi-objective genetic programming," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 122–132.
[78] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and M. Kessentini, "On the impact of refactoring on the relationship between quality attributes and design metrics," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–11.

Chaima Abid is currently a PhD student in the intelligent Software Engineering group at the University of Michigan. Her PhD project is concerned with the application of intelligent search and machine learning in different areas such as web services, refactoring, mobile app reviews and security. Her current research interests are Search-Based Software Engineering, web services, refactoring, security, data analytics and software quality.



Dhia Elhaq Rzig is currently a Phd Student in the Intelligent Software Engineering group at the University of Michigan Dearborn, His PhD work and research interest concern the intersection of Machine Learning and Software engineering, such as the adoption of DevOps within ML projects workflows', using AI and intelligent search to improve Software Refactoring, among other Topics. He obtained his Master's in Information Processing from a joint program of the University of Paris Descartes and the National Engineering School of Tunis, and his Bachelor's in Software Engineering from the National Institute of Applied Sciences and Technology in Tunisia.

Thiago do Nascimento Ferreira is currently a Postdoctoral Researcher at University of Michigan-Dearborn in USA. He received the PhD degree in Computer Science from the Federal University of Paraná in Brazil, in 2019. His main interest are bio-inspired computation, multi-objective optimization and preference-based optimization algorithms focused on Search-based Software Engineering (SBSE).

Marouane Kessentini is a recipient of the prestigious 2018 President of Tunisia distinguished research award, the University distinguished teaching award, the University distinguished digital education award, the College of Engineering and Computer Science distinguished research award, 4 best paper awards, and his AI-based software refactoring invention, licensed and deployed by industrial partners, is selected as one of the Top 8 inventions at the University of Michigan for 2018 (including the three campuses), among over 500 inventions, by the UM Technology Transfer Office. He is currently a tenured associate professor and leading a research group on Software Engineering Intelligence. Prior to joining UM in 2013, He received his Ph.D. from the University of Montreal in Canada in 2012. He received several grants from both industry and federal agencies and published over 110 papers in top journals and conferences. He has several collaborations with industry on the use of computational search, machine learning and evolutionary algorithms to address software engineering and services computing problems.

Tushar Sharma Tushar Sharma is currently a research scientist at Siemens Corporate Technology, Charlotte, USA. He earned PhD from Athens University of Economics and Business, Athens, Greece with specialization in software engineering in May 2019. Earlier, he obtained an MS in Computer Science from the Indian Institute of Technology-Madras, Chennai, India. His professional experience includes working with Siemens Research and Technology Center, Bangalore, India for more than seven years (2008-2015). He co-authored the book Refactoring for Software Design Smells: Managing Technical Debt and two Oracle Java certification books. He has founded and developed Designite which is a software design quality assessment tool used by many practitioners and researchers worldwide. He is an IEEE Senior Member.