X-SBR: On the Use of the History of Refactorings for Explainable Search-Based Refactoring and Intelligent Change Operators

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 an enhanced knowledge-informed multi-objective search algorithm, called X-SBR, to provide explanations for refactoring solutions and improve the generated recommendations. 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**

I. 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 that is more adequate for continuous problems [21]. In fact, a 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 depending to 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 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 [22] to generate association rules using the refactoring history and quality analysis of 18 projects of different sizes and categories. We used RMiner [23] to detect the refactoring operations performed between the commits. 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 attributes/fitness functions (e.g. extendibility, functionality, flexibility etc.). Thus, these patterns 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.

We evaluated the execution time, quality of refactoring recommendations and identified refactoring patterns using different evaluation metrics. Statistical analysis of our experiments based on 4 open source systems showed that our proposal performed significantly better than four existing search-based refactoring approaches [9], [11], [24], [25] and an existing refactoring tool not based on heuristic search, JDeodorant [26], in terms of improving the quality and enhance the trustworthiness to apply the recommended refactorings. We used these 5 refactoring tools and the 4 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.

Replication Package. All material and data used in our study are available in our replication package [27].

II. X-SBR Approach

A. Overview

The goals of this paper are to 1) develop a knowledge-informed NSGA-II [28] 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 includes the following steps.

Step 1: Data collection and static analysis of the code to extract refactoring operations and quality metrics.

Step 2: Generation of association rules to link quality metrics with refactoring operations from the data 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 attributes (refer to Table II) and we extracted the refactoring using RMiner [23]. Based on the collected data, we applied the Apriori algorithm to find association rules to link low-level quality attributes and refactoring operations with high-level quality attributes. 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.

B. Training Data

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 attributes (described in Table II) that are computed using a set of eleven weighted object-oriented design properties (described in Table I). We selected this model as it has been extensively used in industry and existing refactoring studies [9], [10], [15], [24], [29]–[31] to assess the quality of software systems. Thus, the paper is not making any new assumptions/validations about QMOOD.

2) Extracting History of Refactorings: In this study, we used RMiner, a tool proposed by Tsantalis et al. [23], 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.

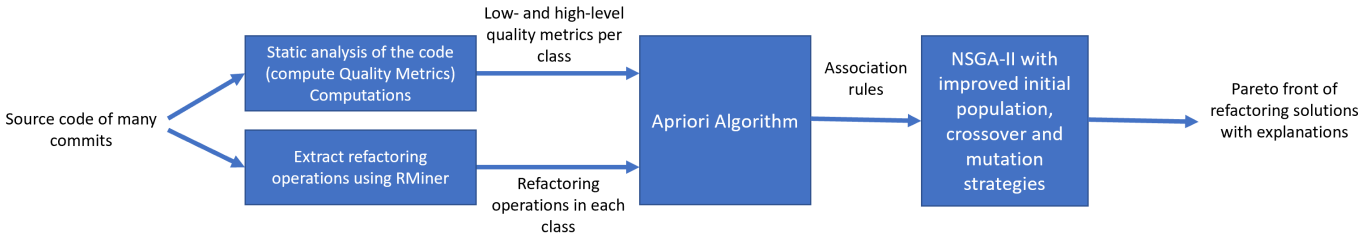

Fig. 1: Approach Overview

TABLE I: 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.

We selected RMiner since it achieved accurate results in detecting refactorings compared to the state of-the-art tools, with a precision of 98% and recall of 87% [23]. We provide in the validation section the details of the collected data related to refactorings and quality metrics on open source projects.

C. Association Rule Mining

Apriori is an algorithm for frequent item-set mining and association rule learning that was first defined by Agrawal et al. [22]. 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 [27]. Before applying the Apriori algorithm, we preprocessed the data by performing discretization. We transformed the continuous design and QMOOD variables 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. 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 right-hand side needs to include only item-sets with elements belonging to the QMOOD metrics.
- The left-hand side needs to have at least 4 elements from the design properties item-set.

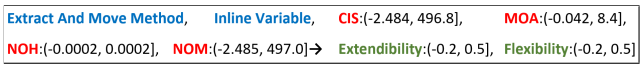

Fig. 2: Example of an association rule

We included 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 total number of attributes in a class—is low. 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

TABLE II: 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
Understandability	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

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. For example, the UI highlighted the metrics contributing to the recommendation of the refactoring and so on.

D. Knowledge-Informed and Explainable NSGA-II for Search-Based Refactoring

1) Proposed Algorithm: NSGA-II [28] 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], [32]. 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 II. 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. It is required to assess the feasibility of solutions and see whether they maintain the behavior of the system using a set of pre- and post-conditions defined by Opdyke [33]. To enable the static analysis of the source code, we used the Soot parser [34] 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. 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) 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. 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

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

Inputs: call graph of a software system P, refactoring operations TC
Output: subset(s) of the refactoring operations
Begin
I:= Instantiation(TC)// vectors of refactoring operations
P_0:=set_of(I)//Population Initialization
t:=0
Repeat
C_t:=apply_Genetic_Operators(P_t)//Apply the genetic operators on population P_t
G_t:=$P_t \cup C_t$ // Combine parent and offspring populations
for all I $\in G_t$ **do**
  Extendibility(I):=calculate_Extendibility(P)
  Effectiveness(I):=calculate_Effectiveness(P)
  Functionality(I):=calculate_Functionality(P)
  Understandability(I):=calculate_Understandability(P)
  Flexibility(I):=calculate_Flexibility(P)
  Reusability(I):=calculate_Reusability(P)
end for
F:=fast_Non_Dominated_Sort(G_t) // F=(F_1,F_2,...), all nondominated fronts of G_t
$P_{t+1} = \emptyset$
i:=1
while $|P_{t+1}| + |F_i| <$ Max_size **do**
  Crowding_distance_assignment(F_i) // calculate crowding distance in F_i
  P_{t+1}= $P_{t+1} \cup F_i$ // include i^{th} nondominated front in parent pop
  i:=i+1
end while
Sort (F_i, $\prec_n$) // sort in descending order using $\prec_n$
P_{t+1}= $P_{t+1} \cup F_i$ $[1...(Max\_size - |P_{t+1}|)]$ // choose the first Max_size - $|P_{t+1}|$ elements of F_i
t:=t+1 // increment generation counter
until t=Max_iteration
best_solutions := first_front(P_t)
return best_solutions

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 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.

3) Crossover: We first start by randomly picking two parents, S_1 and S_2, from the current population. 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.

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.

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 . 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.
- We deduce the refactorings that are not associated with any pattern.
- We look for the rules that improve the weakest objective of S.
- 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.
- 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.

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 [35]. 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 [23]. 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.

III. Experiment and results

A. 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?*

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. We performed static analysis on the code to compute low- and high-level code quality metrics. Then, we used RMiner [23] to detect the refactoring operations performed between the commits. Our dataset can be found in the appendix website [27]. After that, we used the Apriori algorithm [22] 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 [24]. 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 III since they are used in

existing refactoring benchmark [24] 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. [24]) which is basically Algorithm 1, but with random initialization, 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. [24]. However, we extended our baseline in RQ3 when evaluating the relevance of the refactoring recommendations.

TABLE III: 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 III. 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 [36]. 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], [24], [25] and also a tool, called JDeodorant [26], not based on heursitic 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.

B. 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 [37].

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) [38]: It measures the proportion of solutions that lie on the reference front RS [39]. The higher this proportion the better is the quality of the solutions.

- *Hypervolume* (I_{HV}) [40]: 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}) [41]: 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. [33]. 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 our previous studies [REF]. The exhaustive list can be found in the online appendix [27].

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.

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

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

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.

$$\text{Manual Correctness} = \frac{|\text{Meaningful Refactorings}|}{|\text{Recommended Refactorings}|} \quad (6)$$

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

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

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

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

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.

C. Parameters tuning

Parameters setting plays an important role in the performance of a search-based algorithm. 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 tested 50, 100, 200, 300 and 500 for the initial population and 1 000, 2 500, 5 000, 10 000 and 100 000 for the maximum number of evaluations. We ended up by choosing 10 000 for the maximum number of evaluations and 100 for the initial population. The crossover and mutation probabilities are set to 0.6 and 0.4 respectively. In order to have significant results, for each couple (algorithm, system), we use the trial and error method in order to obtain a good parameter configuration. Trial and error is a fundamental method of problem solving. It is characterized by repeated and varied attempts of algorithm configurations. Thus, a reasonable set of parameter values have been studied.

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 [42] 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 [43].

We found that all the results above 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) 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 IV 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.

D. Subjects

We selected 14 participants to evaluate *X-SBR* on the 4 systems described in Table III. We carefully selected them to make sure that they extensively applied refactorings during their

TABLE IV: 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

previous experiences in development and also used the open source systems extensively in their previous and current projects in industry. They 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 V (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 [44]. 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.

E. Results

1) Results for RQ1: 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

TABLE V: 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

TABLE VI: 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

of a rule created by the Apriori algorithm. The complete list can be found in our online appendix [27]. Table VI 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. [24]), (2) NSGA-II with an improved initial population strategy, (3) NSGA-II with improved change operators, and (4) NSGA-II with improved change operators and initial population strategy (*X-SBR*). Figure 3 shows the average time spent to run the four systems 30 times. In all four projects, the traditional NSGA-II [24] has the lowest execution time which is expected since the initialization and change operators are done randomly. In all other three algorithms, the program needs some execution time to access and query the large database of rules. However, the difference in execution time is negligible and does not exceed a few seconds. Therefore, it would be a very small sacrifice if we get better high-quality solutions. It is interesting to note that, for Apache Ant and ArgoUML, the average execution time was relatively high when we used the random initialization and the improved change operators and was reduced significantly when we improved the seeding strategy. This leads us to the hypothesis that, in our proposed approach, the initialization process has a bigger impact on the convergence of the algorithm than the change operators. Table VII 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 VIII contains the results of the Contribution (I_C) metric of the three modified algorithms compared to the traditional NSGA-II [24]. 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

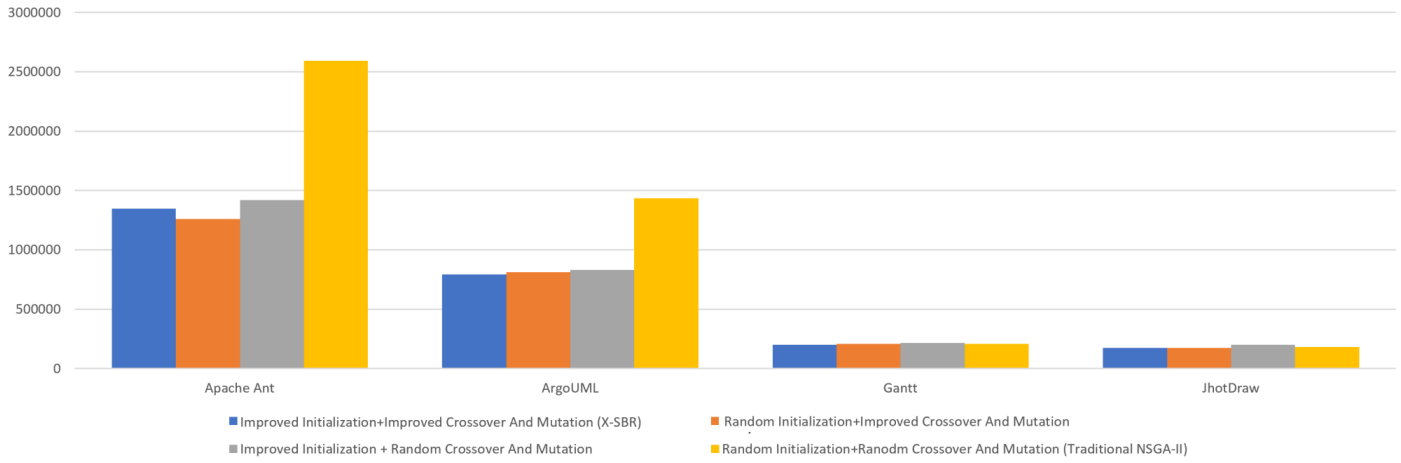

Fig. 3: Average execution time (ms) of all algorithms using the four systems

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 limited 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 traditional NSGA-II [24] demonstrates better diversity and execution time than *X-SBR* but the difference is small for both metrics

2) Results for RQ2: Figure 4 shows the average number of invalid refactorings in the solutions of the Pareto front in all four systems using the different algorithms. 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 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.

3) Results for RQ3: We summarize in the following the feedback of the developers based on the survey. Figure 5 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 [24] 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. [24] 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 6 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

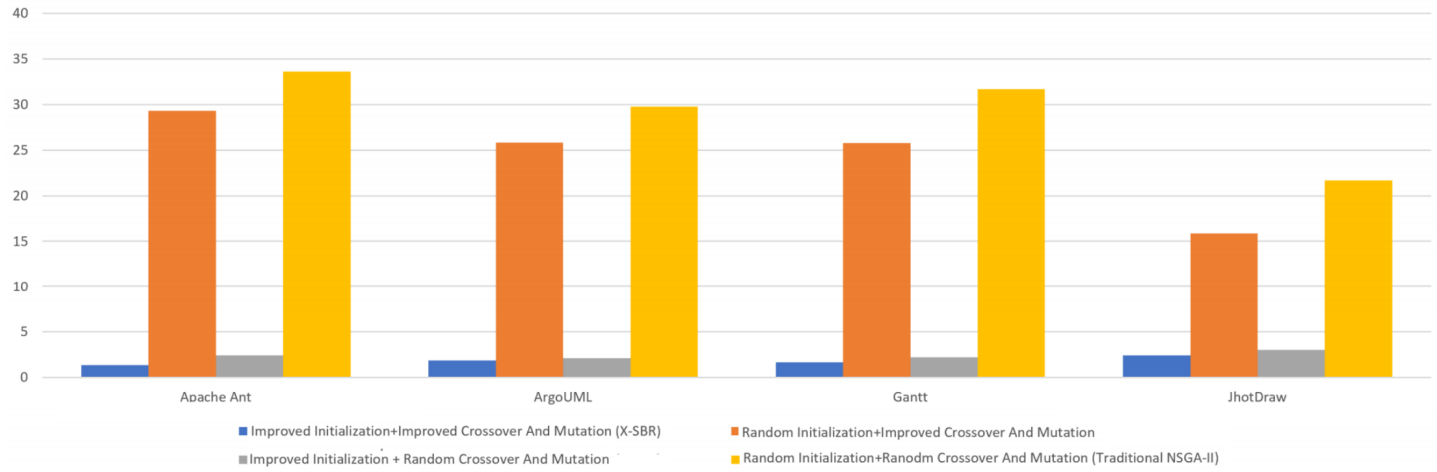

Fig. 4: Average number of invalid refactorings in the solutions of all algorithms using the four systems

TABLE VII: 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

TABLE VIII: 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

which boosted their trust in the refactoring tool.

IV. 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 [45].

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 adapted a counter-balanced design.

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. Almost all of our industrial collaborators in the refactoring area are selecting major refactoring strategies based on discussions between the architects to adopt the best alternative. For the selection threat, the participant diversity in terms of experience could affect the results of our study. We addressed the selection threat by giving a lecture and tests.

External threats. 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.

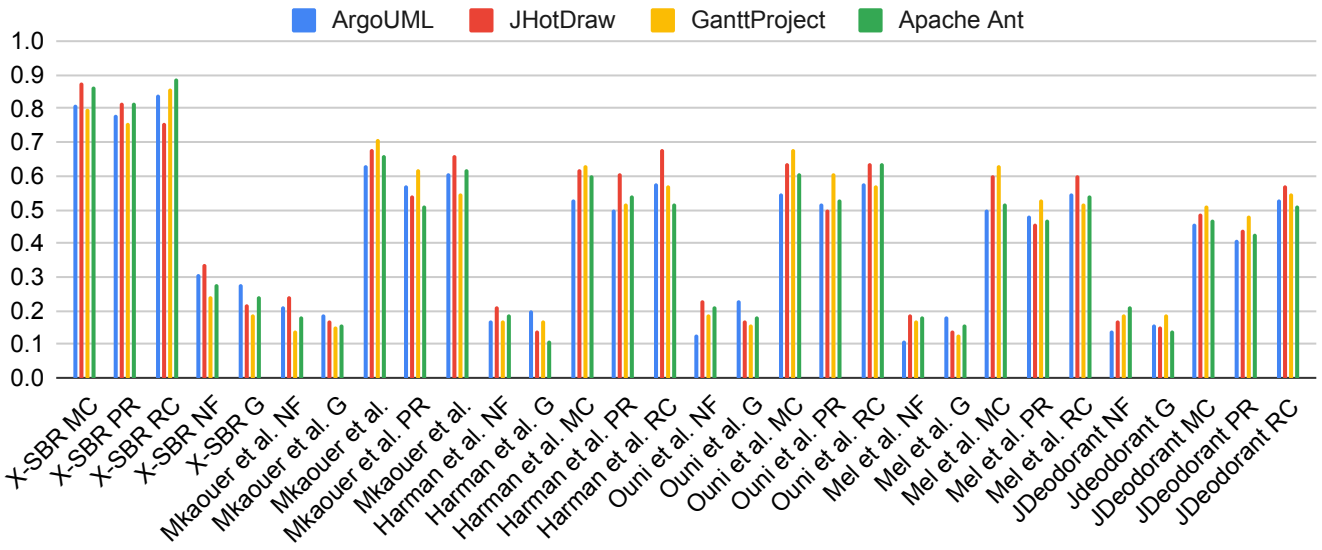

Fig. 5: Automated and manual evaluation of refactoring recommendations generated by the different refactoring tools

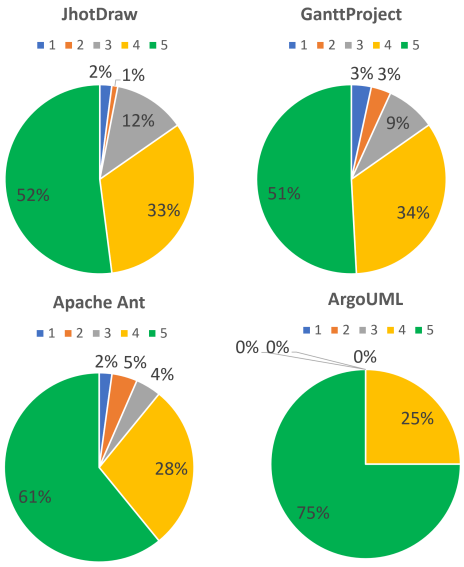

Fig. 6: Distribution of the relevance of the explanations according to the survey results (1=not relevant-5=very relevant)

V. Related Work

Many studies have used search-based techniques to automate software refactoring by optimizing different sets of quality metrics [9], [11], [17], [24], [25], [31]. 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].

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. [31] 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. [46] 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.

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 are more adequate for continuous problems and 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.

VI. Conclusion

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 4 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.

In our future work, we will add other fine-grained refactoring operations, such as Decompose Conditional, Replace Conditional with Polymorphism, and Replace Type Code with State/Strategy. Another future research direction related to our work is to include code smell history and bug reports when generating association rules.

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] S. M. Elsayed, R. A. Sarker, and D. L. Essam, "Ga with a new multi-parent crossover for solving ieee-cec2011 competition problems," in *2011 IEEE congress of evolutionary computation (CEC)*. IEEE, 2011, pp. 1034–1040.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] W. F. Opdyke, "Refactoring object-oriented frameworks," 1992.
[34] 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.
[35] E. R. Murphy-Hill and A. P. Black, "Why don't people use refactoring tools?" in *WRT*, 2007, pp. 60–61.
[36] 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.
[37] 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.
[38] 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.
[39] 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.
[40] 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.
[41] D. A. Van Veldhuizen and G. B. Lamont, "Multiobjective evolutionary algorithm research: A history and analysis," Citeseer, Tech. Rep., 1998.
[42] 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.
[43] 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.
[44] 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.
[45] 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.
[46] 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.