Intelligent Change Operators for Search Based Refactoring

Chaima Abid[1], James Ivers[2], Thiago do N. Ferreira[1], Marouane Kessentini[1], Fares E. Kahla[1] and Ipek Ozkaya[2]
[1]University of Michigan-Dearborn, Dearborn, USA
[2]Carnegie Mellon University Software Engineering Institute, Pittsburgh, USA
{cabid, thiagod, marouane}@umich.edu, {jivers, ozkaya}@sei.cmu.edu and benkahlafares@gmail.com

***Abstract*—In this paper, we propose intelligent change operators and integrate them into an evolutionary multi-objective search algorithm to recommend valid refactorings that address conflicting quality objectives such as understandability and effectiveness. The proposed intelligent crossover and mutation operators incorporate refactoring dependencies to avoid creating invalid refactorings or invalidating existing refactorings. Further, the intelligent crossover operator is augmented to create offspring that improve solution quality by exchanging blocks of valid refactorings that improve a solution's weakest objectives. We used our intelligent change operators to generate refactoring recommendations for four widely used open-source projects. The results show that our intelligent change operators improve the diversity of solutions. Diversity is important in genetic algorithms because crossing over a homogeneous population does not yield new solutions. Given the inherent nature of design trade-offs in software, giving developers choices that reflect these trade-offs is important. Higher diversity makes better use of developers time than lots of incredibly similar solutions. Our intelligent change operators also accelerate solution convergence to a feasible solution that optimizes the trade-off between the conflicting quality objectives. Finally, they reduce the number of invalid refactorings by up to 71.52% compared to existing search-based refactoring approaches, and increase the quality of the solutions. Our approach outperformed the state-of-the-art search-based refactoring approaches and an existing deterministic refactoring tool based on manual validation by developers with an average manual correctness, precision and recall of 0.89, 0.82, and 0.87.**

***Index Terms*—refactoring dependencies, intelligent change operators, multi-objective refactoring recommendation.**

I. Introduction

Even for the most competent organizations, building and maintaining high performing software applications with high quality is a challenging and expensive endeavor [55]. Working in fast-paced environments that demand frequent releases across several products and deployment environments often forces developers to compromise high quality standards in favor of meeting deadlines [33]. As software systems continue to grow in size and complexity, their maintenance continues to become more challenging and costly [24], [13]. To improve the quality and maintainability of software systems, developers take advantage of refactoring as a means to improve the structure of code without affecting its external behavior [18].

Manual refactoring is generally a labor-intensive, ad hoc, and potentially error-prone process [42]. To improve this gap, a wide range of work has focused on automating refactoring recommendations using a variety of techniques that include template/rule-based tools [54], [53], static and lexical analysis [9], [14], and search-based software engineering [35]. Recent surveys show that search-based software engineering has been increasingly used to find refactoring recommendations [41], [35] to address the trade-offs among conflicting quality metrics and the large search space of potential refactoring strategies. For instance, O'Keeffe et al. [52] compared different local search-based algorithms such as hill climbing and simulated annealing to generate refactoring recommendations that improve static quality metrics [8]. Harman et al. [23] proposed using multi-objective search for refactorings that improve coupling and reduce cohesion. Ouni et al. [50] and Mkaouer et al. [40] proposed multi-objective and many-objective techniques to balance conflicting quality metrics when finding refactoring recommendations. Hall et al. [21] and Alizadeh et al. [4] improved the state-of-the-art of search-based refactoring by enabling interaction with developers and learning their preferences. More detailed descriptions of existing search-based refactoring studies can be found in the following surveys [35], [41]. Despite the promising results of search-based refactoring on both open-source and industry projects, several limitations that reduce their effectiveness remain unaddressed. While these limitations apply, in general, in most applications of search-based reasoning approaches to software engineering problems [2], [22], [51], we focus on search-based refactoring in this paper.

Existing refactoring recommendation tools, including those that use non-search-based approaches, routinely generate solutions that include invalid refactorings because they do not account for dependencies among refactorings. Manually applying a sequence of refactorings is common practice in existing tools [11], [42], [10], however these tools treat each refactoring in the sequence in isolation. For instance, Cinnéide et al. [43] investigated the impact only of individual refactorings on quality attribute metrics, such as using Move Method to reduce the coupling of a class, without studying the impact of a sequence of refactorings. Figure 1 shows an example of the refactoring recommendations generated by JDeodorant [57] where, similar to other refactoring recommendation tools, the dependencies between the refactorings are not apparent, thus leaving the challenging task of dealing with invalid refactorings to developers. Consequently, developers often prefer manually applying refactorings to using such tools. A key contributor to this problem is that search-based refactoring

approaches employ random change operators (e.g., crossover and mutation) to evolve solutions without considering the dependencies among refactorings. Without detecting which refactoring dependencies exist, the change operators used by algorithms routinely invalidate solutions by breaking refactoring dependencies or introducing refactorings whose dependencies are not satisfied. Furthermore, refactoring dependencies provide clues that could be exploited in more intelligent crossover operations to improve decisions on which part(s) of solutions to exchange to produce higher quality offspring.

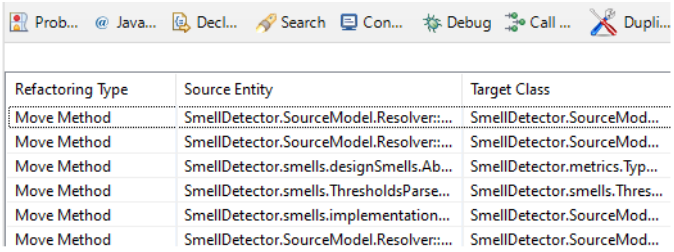


Fig. 1: Sample refactoring recommendations from JDeodorant.

In this paper, we propose intelligent change operators and integrate them into a multi-objective search algorithm, based on NSGA-II [15], to recommend valid refactorings that address conflicting quality objectives such as Reusability, understandability, and effectiveness. The proposed intelligent crossover and mutation operators use: i) the dependencies detected among refactorings to decompose a solution into blocks of refactorings; and ii) the effects of these blocks on objectives to identify good genes from parents to generate high-quality offspring. A refactoring dependency exists when one refactoring cannot be successfully applied without first applying another. Partitioning refactorings into blocks such that no dependencies span blocks allows change operators to use blocks as the unit of change to avoid invalidating refactorings. Our tool calculates the effect of each block within a solution on the objectives and uses this data to select which blocks to exchange between solutions to improve the first solution's weaknesses (e.g. the objective with smallest values).

We applied our intelligent change operators to generate refactoring recommendations for four widely used open-source projects and compared this approach to five existing refactoring techniques in terms of the diversity of the solutions, number of invalid refactorings, and the quality of generated solutions. We also conducted a survey with 14 developers to evaluate the correctness and relevance of the refactorings generated by the different algorithms for these projects.

The results show that our technique performed significantly better than the four existing search-based refactoring approaches [39], [23], [52], [48] and an existing refactoring tool not based on heuristic search, JDeodorant [56], with an average manual correctness, precision and recall of 0.89, 0.82, and 0.87, respectively. We used these five refactoring tools and open source projects because: i) they are representative of automated multi-objective search-based refactoring recommendation techniques; ii) they are publicly available (including the non search-based tool); and iii) the familiarity of the participants with these open source systems.

Replication Package. All material and data used in our study are available in our replication package [7].

II. Dependency-Aware Refactoring Recommendation System

A. Background: Multi-Objective Refactoring Using NSGA-II

Multi-objective optimization has been widely applied to refactoring problems to find trade-offs when searching for solutions. Non-Dominated Sorting Genetic Algorithm II (NSGA-II) [15] (Algorithm 1) is the dominant multi-objective optimization algorithm that has been used in search-based software engineering, including for search-based refactoring [23], [52], [50], [3], [4], [40]. NSGA-II is designed to find a set of non-dominated solutions (a Pareto-front) in which each solution is a sequence of refactorings that provides a compromise among conflicting objectives (e.g., quality metrics).

Algorithm 1: NSGA-II algorithm.

```
Input: System to evaluate and list of refactoring types
Output: Non-dominated refactoring solutions
Generate a random population P and evaluate the objectives;
while the stopping condition is not reached do
    Select individuals M from P using Binary Tournament
      Selection;
    Apply crossover operation on M to generate the offspring
      population O;
    Apply mutation operation on O;
    Update P by combining the parent and offspring populations;
end
return P;
```

Initially, a starting population P is created using a random procedure. These solutions then undergo crossover and mutation, producing offspring O, and the process is repeated until the stopping condition is reached (in our case, a maximum number of generations). The objective values of the solutions are computed and change operators are applied to create the next generation. In most of existing adaptations, including this paper, the algorithm finds non-dominated solutions balancing several conflicting objectives, such as the six QMOOD quality metrics [8]. The different objectives can be normalized if they have different scales. Each objective can be written as follow:

$$Objective_i = \frac{Q_i^{after} - Q_i^{before}}{Q_i^{before}} \quad (1)$$

where Q_i^{before} and Q_i^{after} are the values of the $quality\_metrics_i$ before and after applying a solution (or sequence of refactorings), respectively.

The search space explored by NSGA-II consists of different refactoring operations applied to different code locations where each operation is represented by a refactoring type (e.g., Move Method) and its parameters (e.g., source class, target class, attributes). In this paper, we selected 14 refactoring types that are frequently used in practice based on existing studies [31], [42], [12]: Encapsulate Field, Decrease Field Security, Decrease Method Security, Increase Field Security,

Increase Method Security, Pull Up Field, Pull Up Method, Push Down Field, Push Down Method, Extract Sub Class, Extract Super Class, Extract Class, Move Field, and Move Method. A vector in which each element represents a refactoring operation is used to represent a solution. Each refactoring operation must satisfy a set of pre- and post-conditions defined by Opdyke [46] to maintain the behavior of the system.

The most common change operators used in search-based refactoring approaches are the random crossover and mutation operators. In these operators, refactorings are selected randomly from solutions for exchange or replacement with others, which can generate invalid refactorings or invalidate other refactorings (e.g., by removing a refactoring another one depends on). We developed three components to improve the change operators used in the NSGA-II algorithm: i) a refactoring dependency detection algorithm; ii) an intelligent crossover that factors in dependency correctness and the implications of collections of refactorings on fitness functions; and iii) a dependency-aware mutation. Finally, we note that the proposed approach, as described later, can be integrated for both NSGA-II and NSGA-III as they are using the same change operators. The difference between them is that NSGA-III uses a set of reference directions (identified via a nitching function), while NSGA-II uses a more adaptive scheme through its crowding distance operator for the same purpose. This difference does not affect our goal of comparing the impact of our intelligent change operators on the final Pareto-front.

B. Refactoring Dependency Theory

Our dependency-aware refactoring recommendation technique relies on an ordering dependency between pairs of refactorings. Specifically, an ***ordering dependency*** ($rf_2 \mapsto rf_1$) between two refactorings (rf_1 and rf_2) exists when rf_2 can only be successfully applied after rf_1 has been applied. That is, rf_1 makes a change to code that is necessary in order to apply rf_2. This condition can be evaluated based on the combination of pre- and post-conditions of the types of refactorings involved and the parameters of each refactoring. For example, to apply Move Method (a type of refactoring) to move method m_1 from class c_1 to class c_2 (m_1, c_1, and c_2 being the parameters of the refactoring), several pre-conditions must hold (e.g., m_1, c_1, and c_2 must all exist and m_1 must be defined on c_1). The pre- and post-conditions of each type of refactoring are described in our online appendix [7] and were extensively validated for correctness and completeness in current literature [49], [44], [17], [36].

Figure 2 shows a simplified example of a refactoring solution that is composed of refactoring operations that depend on each other. Three of the refactorings (#3, #4, #5) depend on another refactoring (#2) because the Extract Super Class refactoring (#2) creates a new class (Client), on which refactorings #3, #4, and #5 operate. If the new class is not created first, then refactorings #3, #4, and #5 will fail. Thus, there exists an ordering dependency from each of #3, #4, #5 to #2.

Refactoring solutions have traditionally been represented as a sequence, likely originating with the common vector

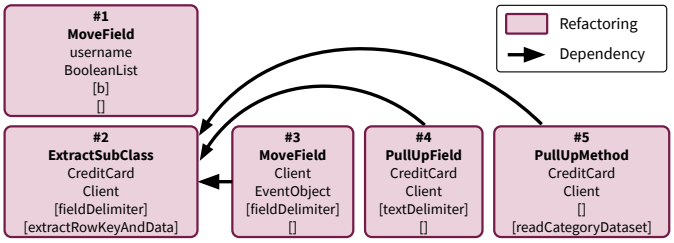


Fig. 2: A simplified example of refactorings that depend on each other.

representation used in many genetic algorithms. In some cases, a solution could be appropriately represented as a set of sequences, but only if the ***refactoring graphs*** are simplistic enough. A ***refactoring graph*** is a weakly connected directed acyclic graph composed of refactoring vertices and ordering dependency edges. In practice, there are many examples where a sequence vs. graph representation is misleading. For example, if two refactorings (rf2 and rf3) both depend on a common refactoring (rf1), we have a graph for which a sequence representation would be misleading. rf1 must precede rf2 and rf3, but there is no dependency between rf2 and rf3. <rf1, rf2, rf3> would be as acceptable as <rf1, rf3, rf2>. A sequence representation indicates an ordering, and the choice of a graph over a sequence allows us to unambiguously indicate only "real" dependencies. As for the initial refactoring sequence, it is true that the order in that sequence does shape the original graphs. However, the initial sequence is generated randomly for each solution in the population, much as if random graphs were generated.

Using the ordering dependencies as the basis for forming refactoring graphs, Algorithm 2 results in a set of graphs with the following traits:

- Each refactoring in a solution is an element of exactly one refactoring graph.
- Some graphs contain a single refactoring because that refactoring is independent of all others. We call these ***trivial graphs***.
- The remaining graphs contain multiple refactorings, each of which is part of one or more dependencies. We call these ***non-trivial graphs***.
- Each refactoring graph is independent of every other graph in the solution.

The dependencies, as described in Algorithm 2, are detected based on comparisons between pre- and post-conditions of refactorings. The algorithm takes a list of refactorings as input and generates a set of refactoring graphs as output.

Line 1 initializes the lists of refactorings (nodes, V) and refactoring dependencies (edges, E). Then, the post-conditions of each refactoring of the solution C (collection of refactorings) are evaluated for matching with the remaining refactorings in C (Lines 2–12). Specifically, the algorithm looks for any match between predicates of pre- and post-conditions. That is, if any predicate of the post-condition of one refactoring (any element of P) matches any predicate of

Algorithm 2: Dependency Detection Algorithm.

Input: Refactoring solution $C = \{r_1, r_2, r_3, \ldots, r_n\}$
Output: Set of refactoring graphs $F = \{f_1, f_2, f_3, \ldots, f_m\}$

```
1  V ← ∅, E ← ∅;
2  foreach r_i ∈ C do
3      V ← V ∪ r_i;
4      P ← post_conditions(r_i);
5      foreach r_j ∈ C | j > i do
6          Q ← pre_conditions(r_j);
7          M ← P ∩ Q;
8          if |M| ≠ 0 then
9              E ← E ∪ {r_j, r_i};
10         end
11     end
12 end
13 G ← (V, E);
14 F ← partition(G);
15 return F
```

the pre-condition of another refactoring (any element of Q), then a dependency has been detected and an edge is added to the graph between those refactorings (Lines 4–11). We repeat this process until all the refactorings have been visited.

C. Proposed Intelligent Change Operators

1) Dependency-aware Crossover: We developed a baseline dependency-aware crossover that only preserves the dependencies among refactorings (e.g., without fixing the weaknesses of refactoring solutions). This version, as shown in Algorithm 3, reduces the occurrence of invalid refactorings in solutions because it preserves refactoring dependencies.

Algorithm 3: Dependency-Aware Crossover Algorithm.

Input: population $S = \{s_1, s_2, s_3, \ldots, s_n\}$ and a probability P
Output: offspring population $S' = \{s_1', s_2', s_3', \ldots, s_n'\}$

```
1  S' ← ∅;
2  for i ← 1 to |S|/2 do
3      {s_a, s_b} ← select random solutions from S;
4      if random_number ≤ P then
5          B_a ← group refactorings of s_a into blocks;
6          B_b ← group refactorings of s_b into blocks;
7          {s'_a, s'_b} ← apply single point crossover on {B_a, B_b};
8          S' ← S' ⋃ {s'_a, s'_b};
9      else
10         S' ← S' ⋃ {s_a, s_b};
11     end
12 end
13 return S';
```

We start by randomly selecting two solutions, s_a and s_b, as parents for new offspring (Line 3). Then, we group the refactorings of s_a and s_b into blocks (Lines 5–6) based on the dependencies detected by Algorithm 2. Each block contains a single trivial or non-trivial graph. We then perform a single-point crossover (Line 7) that exchanges blocks of refactorings rather than individual refactorings, which avoids invalidating refactorings because all dependencies are isolated within blocks. This results in two offspring, each with genetic information from both parents.

2) Intelligent Crossover: Our intelligent crossover operator is an improvement over random crossover in two ways: it uses refactoring dependencies to reduce the occurrence of invalid refactorings and it chooses blocks of refactorings for exchange that will improve a solution's weaknesses, producing higher quality offspring. The pseudo-code of our proposed intelligent crossover operator is presented in Algorithm 4.

Algorithm 4: Intelligent Crossover Algorithm.

Input: Population $S = \{s_1, s_2, s_3, \ldots, s_n\}$ and a probability P
Output: Offspring population $S' = \{s_1', s_2', s_3', \ldots, s_n'\}$

```
1  S' ← ∅;
2  for i ← 1 to |S|/2 do
3      {s_a, s_b} ← select random solutions from S;
4      if random_number ≤ P then
5          s_best ← higher quality solution of s_a and s_b;
6          s_worst ← lower quality solution of s_a and s_b;
7          B_best ← group refactorings of s_best into blocks;
8          B_worst ← group refactorings of s_worst into blocks;
9          W_best ← get all weaknesses of s_best;
10         if W_best = ∅ then
11             W_best ← get the objective that improves the least
                with s_best;
12         end
13         I ← sort the blocks of B_worst based on potential
              improvement to S_best;
14         I' ← select the blocks from I that improve S_best;
15         n ← select random number between 0 and |I'|;
16         {s'_best, s'_worst} ← apply single point crossover,
              exchanging n blocks between B_best and I;
17         S' ← S' ⋃ {s'_best, s'_worst};
18     else
19         S' ← S' ⋃ {s_a, s_b};
20     end
21 end
22 return S';
```

In essence, the intelligent crossover operator mixes the best genes of the weaker solution with random genes of the better solution (Figure 3). First, we randomly select two solutions, s_a and s_b (Line 3). We then determine the better solution by computing how much each solution improves the objectives (using a weighted sum) of the project to be refactored (Lines 5–6). As before, we group refactorings of both solutions into blocks (Lines 7–8) to preserve refactoring dependencies during crossover. We then determine which objectives are considered the weaknesses of the better solution S_{best} (Lines 9–12) (part a in Figure 3). Any objectives that are worse after applying the better solution are considered weaknesses (e.g. objective 2 in Figure 3 part a). If no objectives are worse after applying the better solution, we select the objective that improves the least after applying the solution as the sole weakness. Then, we sort the blocks of the weaker solution B_{worst} based on how each would impact the objectives (using a weighted sum) of the better solution S_{best} (Line 13) (part b in Figure 3). In part c of Algorithm 4, We pick a random number between 1 and the number of blocks in the weaker solution that would improve the better solution (Line 15) to determine the number of blocks for crossover. Finally, we create two offspring using single point crossover (Line 16) that moves the n blocks from the weaker solution with the best impact on the stronger solution's

objectives to the stronger solution and n random blocks from the stronger solution to the weaker solution.

3) Dependency-aware Mutation: Our proposed dependency-aware mutation operator is defined in Algorithm 5 and illustrated in Figure 4. We modified the random mutation operator to preserve refactoring dependencies. For each solution S, we randomly select a floating-point value. If this value is less than the mutation probability (Line 1), we detect refactoring dependencies (Part a in Figure 4) and identify mutable refactorings (Line 2) (Part b in Figure 4). A mutable refactoring must satisfy at least one of the following:

- it does not participate in any dependencies (e.g., E and C in Figure 4).
- it is part of a non-trivial graph, but no other refactorings depend on it (e.g., G, I and H in Figure 4).
- it is part of a non-trivial graph, but it has an unsatisfied pre-condition and is already invalid (e.g., A in Figure 4).

Then, we chose a random number between 1 and the number of mutable refactorings (Line 3). This number represents the number of refactorings that we will mutate in refactoring solution S. Finally, we replace N refactorings in S with random refactoring operations and parameters (Line 4–7) (Part c in Figure 4).

Algorithm 5: Dependency-aware Mutation Algorithm.

```
   Input: Solution S = {r1, r2, r3, ..., rn} and a probability P
   Output: Mutated solution S
1  if random_number ≤ P then
2      M ← detect mutable refactorings from S;
3      N ← random number between 1 and |M|;
4      for i ← 0 to N by 1 do
5          r_j ← random refactoring from M;
6          replace r_j in S with a random refactoring;
7      end
8  end
9  return S;
```

III. Empirical Study

A. Research Questions

The following research questions guide the evaluation of our proposed approach:

RQ1. Correctness. To what extent can our approach reduce the number of invalid refactorings compared to other multi-objective refactoring recommendation techniques?

RQ2. Quality. To what extent can our approach generate refactoring solutions with better diversity, convergence, and quality improvement compared to other multi-objective refactoring techniques?

RQ3. Relevance. How do developers evaluate the impact of our approach in practice?

To answer item **RQ1**, we chose the algorithm proposed by Mkaouer et al. [39] based on NSGA-III, because it outperforms the existing multi-objective techniques [23], [52], [48] that use random change operators. Please note that NSGA-II and NSGA-III are using the same change operators as explained in the previous section. We also considered two operation-variants of NSGA-II that optimize the same quality objectives as summarized in Table I.

TABLE I: The three operation-variants of the NSGA-II algorithm.

Algorithm	Definition
NSGA-II	NSGA-II with random Single Point crossover and Bit Flip mutation (Mkaouer et al. [39])
Dep-NSGA-II	NSGA-II with dependency-aware change operators (Sections II-C1 and II-C3)
Intel-NSGA-II	NSGA-II with intelligent crossover and dependency-aware mutation (Sections II-C2 and II-C3)

We selected four open-source Java projects (show in Table II) that were used in the work of Mkaouer et al. [39]. These projects are from different domains and have different sizes along with a significant number of contributors over more than 10 years. Furthermore, the selected projects are widely used and extensively involved over time which may justify the need for refactoring.

Also, we checked the validity of pre- and post-conditions of all refactorings in all solutions in each generation for all three algorithms on the four projects. We measured the total number of conflicts for each generation as the percentage of invalid refactorings among all refactorings in all solutions in that generation. We also measured the percentage of invalid refactorings per solution in each generation to see the distribution of invalid refactorings across solutions.

TABLE II: Open-source projects studied.

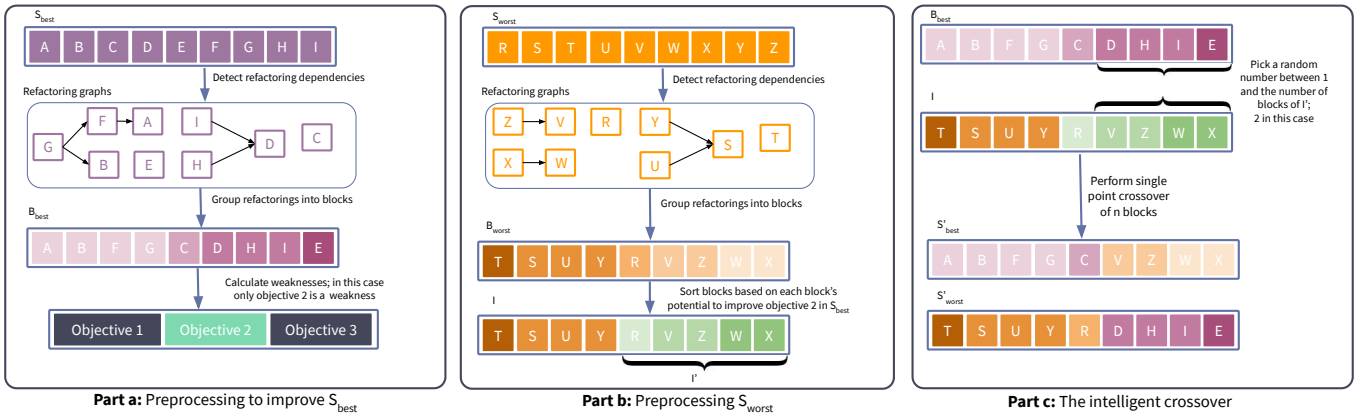
System	Release	# of Classes	KLOC
ArgoUML	v0.3	1358	114
JHotDraw	v7.5.1	585	25
GanttProject	v1.11.1	245	49
Apache Ant	v1.8.2	1191	112

To answer item **RQ2**, we compared the three algorithms in terms of execution time, performance indicators, and improvement in quality metrics of the Pareto-front solutions. Due to the stochastic and non-deterministic nature of meta-heuristic algorithms, different runs of the same algorithm solving the same problem typically give different outcomes. For this reason, we performed 30 runs for each algorithm on each project to make sure that the results are statistically significant.

Finally, to answer item **RQ3**, we conducted a survey with a group of 14 active developers to identify and manually evaluate the relevance of the refactorings generated by our approach. At the top of the criteria mentioned above, the projects used for answering item **RQ1** were selected since the participants are familiar with them so they can provide relevant feedback given their knowledge.

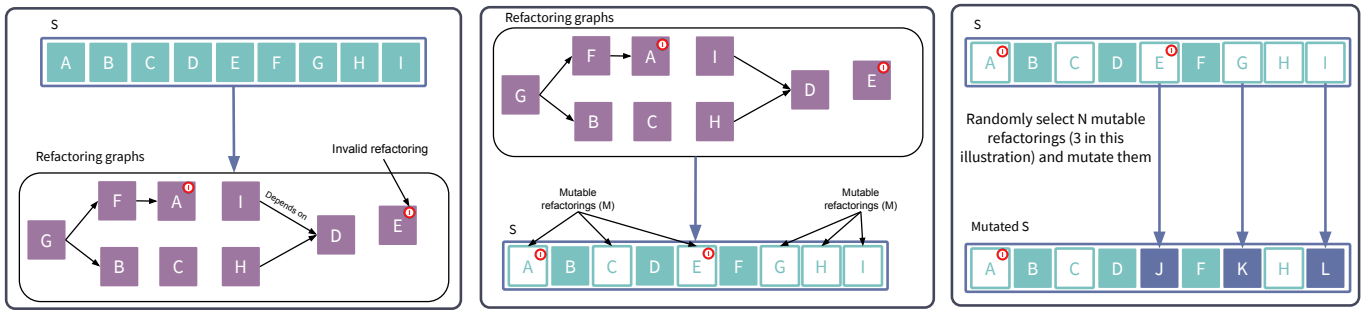
B. Evaluation Metrics

We validate our results using the following metrics.



Part a: Preprocessing to improve S_{best} **Part b:** Preprocessing S_{worst} **Part c:** The intelligent crossover

Fig. 3: An illustration of the intelligent crossover.



Part a: Detect refactoring dependencies **Part b:** Detect mutable refactorings **Part c:** Mutate random mutable refactorings

Fig. 4: An illustration of the dependency-aware mutation.

For item **RQ1**, we want to estimate the correctness of the solutions generated by the three algorithms. For that, we compute the percentage of invalid refactorings in each generation by inspecting the validity of pre- and post-conditions of each refactoring operation. These conditions are discussed by Opdyke et al. [46]. The exhaustive list can be found in the online appendix [7]. We also computed the percentage of invalid refactorings per refactoring solution generated by the three algorithms at each generation.

For item **RQ2**, we use the following three metrics as performance indicators to evaluate the quality of solutions generated by the three algorithms:

- *Contributions* (I_C) [16] measures the proportion of solutions that lie on the reference front (RS) [38]. The higher this proportion, the better the quality of solutions.
- *Inverted Generational Distance* (I_{GD}) [58] is a convergence measure that corresponds to the average Euclidean distance between the approximate Pareto-front provided by an algorithm and the reference Pareto-front. Small values are desirable.
- *Hypervolume* (I_{HV}) [62] measures the volume covered by members of a Pareto-front in objective space delimited by a reference point. An important feature of this metric is its ability to capture diversity and convergence of solutions. A higher hypervolume value is desirable.

We also calculated another metric based on QMOOD that estimates the quality improvement for the project by comparing the quality before and after refactorings generated by the three algorithms. For each refactoring solution S, the quality improvement after applying S is estimated as:

$$Q_S = \sum_{i=1}^{6} Q_{q_i} \ where \ Q_{q_i} = q_i' - q_i \tag{2}$$

where q_i and q_i' represent the value of QMOOD quality attribute i before and after applying S, respectively. For each algorithm, we average the normalized quality improvements across solutions in the Pareto-front generated by each algorithm and we compare them. In addition, we compute the execution time of each generation using the three algorithms.

Finally, for item **RQ3**, we validated the generated refactoring solutions quantitatively and qualitatively. For qualitative assessment, we compared our solutions to a baseline of solutions generated by other multi-objective techniques [23], [52], [48], [39] and by JDeodorant [56], a tool not based on heuristic search. All the search-based refactoring techniques are based on multi-objective search, but each uses different objectives and solution representations. All use the same random change operators, which helps to confirm whether good recommendations result from using our intelligent change operators. The

current Eclipse plug-in version of JDeodorant identifies some types of design defects using quality metrics and proposes a list of refactorings to fix them. For the comparison with JDeodorant, we limited the comparison to the same refactoring types supported by both our approach and JDeodorant. For the quantitative assessment, we calculated precision and recall scores by comparing the refactorings recommended by each of the multi-objective algorithms and JDeodorant with those refactoring manually suggested by the participants (the expected refactorings).

$$Precision = \frac{\text{Recommended Refactorings} \cap \text{Expected Refactorings}}{\text{Recommended Refactorings}} \tag{3}$$

$$Recall = \frac{\text{Recommended Refactorings} \cap \text{Expected Refactorings}}{\text{Expected Refactorings}} \tag{4}$$

After the developers manually suggested refactorings for the projects, we asked them to evaluate the tools' recommendations since their suggestions may not be the only reasonable solution. We asked the participants to assign 0 or 1 to every refactoring solutions generated by the multi-objective algorithms and JDeodorant. A 0 means that the refactoring is not relevant or invalid, and 1 means that the refactoring is meaningful and relevant.

We computed manual correctness as the number of meaningful refactorings divided by the total number of recommended refactorings. Meaningful refactorings were identified by considering the majority opinion across participants for each refactoring.

$$\text{Manual Correctness} = \frac{|\text{Meaningful Refactorings}|}{|\text{Recommended Refactorings}|} \tag{5}$$

C. Parameters Tuning

In order to fairly compare the results among the three algorithms in Table I and the multi-objective algorithms used in our survey [23], [52], [48], [39], we performed the same number of evaluations per run (3k) and used the same initial population size (100). We used the maximum number of evaluations as our stopping criterion. The crossover and mutation probabilities are set to 0.95 and 0.02 respectively. The minimum and maximum number of refactorings per solutions are set to 100 and 200, respectively.

D. Subjects

We evaluated our approach with 14 active industry developers who volunteered to participate in our survey as part of an industry-sponsored research collaboration. We selected individuals with extensive experience applying refactorings in industry and using the selected open source projects in their work. Each filled out a pre-study survey that collects background information, such as their programming experience and their role within their companies.

We divided the participants into four groups balancing skill level and familiarity with the open source projects. The details of the participants and the projects they evaluated are found in Table III. We gave participants a two-hour lecture about software quality assessment and refactoring. During the two-hour lecture, we did not reveal to the participants which refactorings were generated by which app to avoid any possible bias. We provided general knowledge regarding refactoring and showed them how to read and interpret the refactoring solutions and focused on explaining the required steps to complete the survey.

We assessed their knowledge on the open source projects and their performance in evaluating and suggesting refactoring solutions. The participants were asked to assess the correctness and relevance of the refactorings recommended by the multi-objective algorithms [23], [52], [48], [39] and JDeodorant [56] on all four projects. They were shown refactoring recommendations per project without knowing where the recommendations came from.

Since the multi-objective algorithms generate many refactoring solutions in the Pareto-front, it was not feasible to ask the participants to evaluate all the solutions. 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 among the objectives, which could be seen as the mono-objective solution with equally weighted objectives if the objectives do not conflict. Thus, we selected the solution with the median hypervolume IHV value. The average number of refactorings evaluated by each participant is 58. We ensured that each refactoring was evaluated by two developers, and we considered it relevant if both agreed (the overall Cohen's kappa was 0.91).

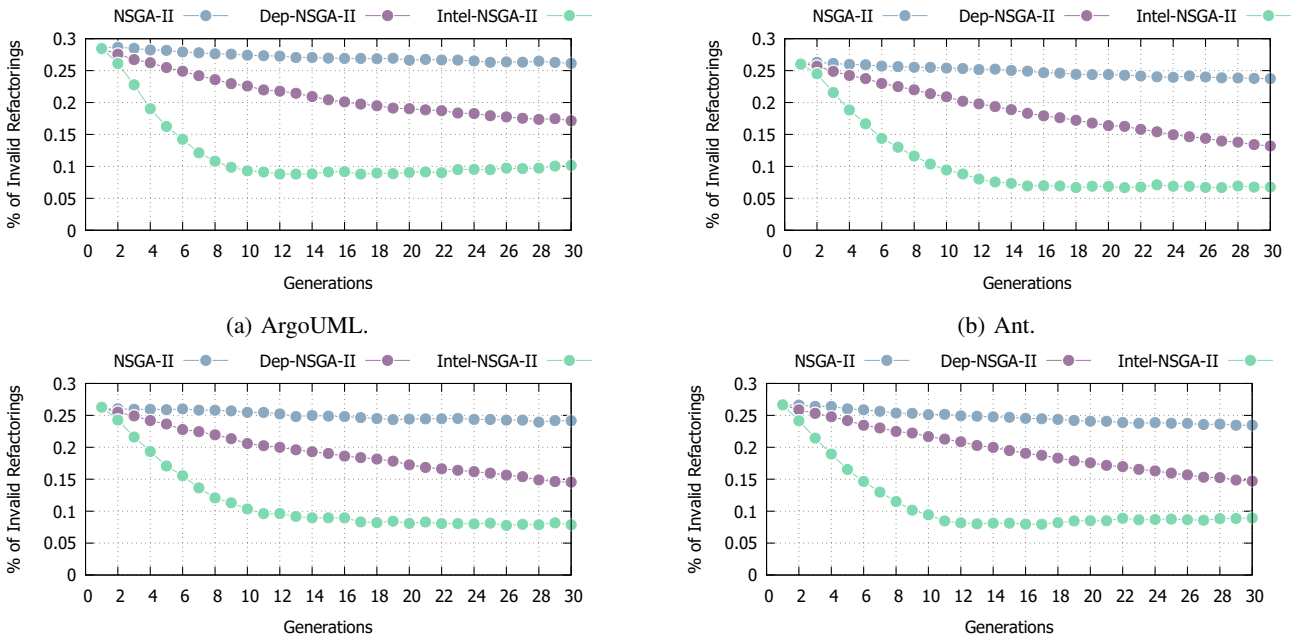
TABLE III: Participant Details.

System	# of Subjects	Avg. Prog. Experience (Years)	Refactoring Experience
ArgoUML	4	10	High
JHotDraw	3	11.5	Very High
GanttProject	3	10.5	High
Apache Ant	4	12	Very High

E. Results

1) RQ1: Correctness: Figure 5 shows the percentage of invalid refactorings across all solutions in each generation for each algorithm for each open source project. All algorithms have 100 non-dominated solutions in the final Pareto-front.

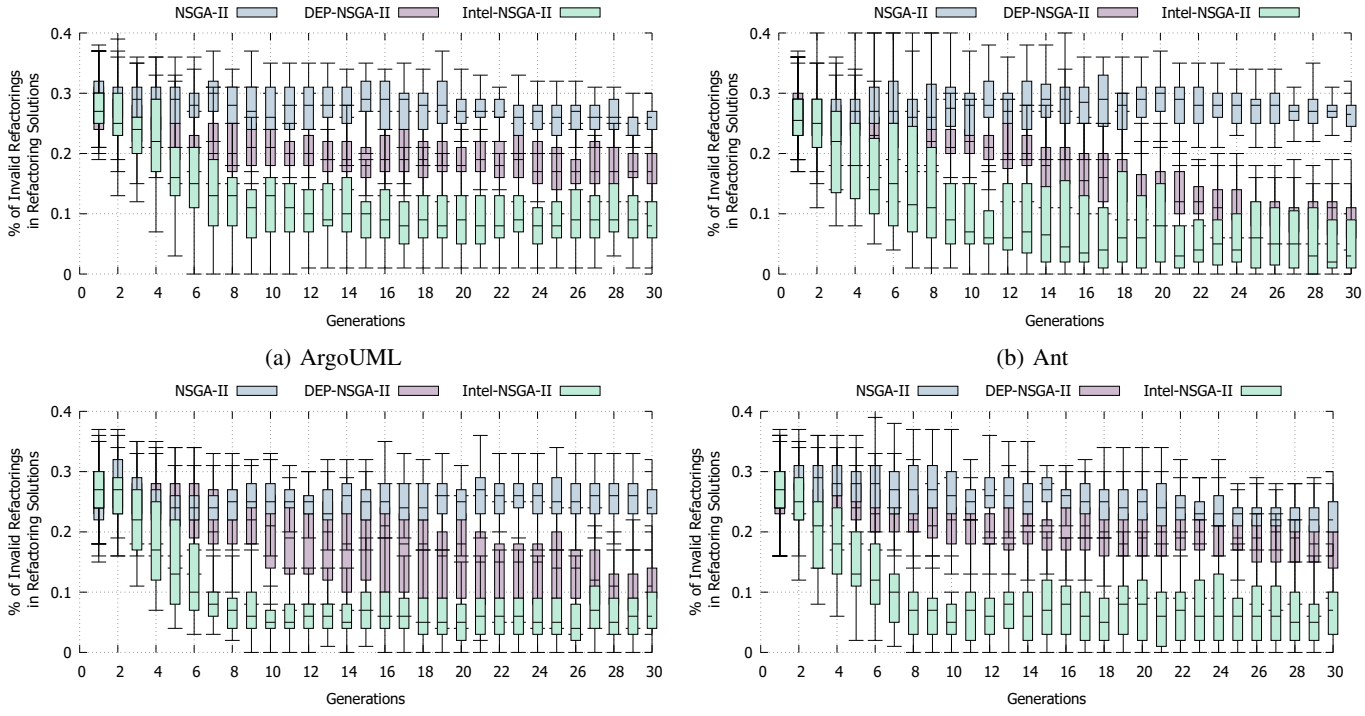
The highest percentages of invalid refactorings for all projects was produced by NSGA-II, though it does reduce the percentage of invalid refactorings by a negligible amount as generations progress. Dep-NSGA-II reduces the percentage of invalid refactorings compared to regular NSGA-II [39] by 44.34%, 34.42%, 39.77%, and 37.29% for Ant, ArgoUML, Gantt, and JHotDraw, respectively. Intel-NSGA-II, however, outperformed the other algorithms and reduces the percentage of invalid refactorings compared to NSGA-II [39] by 71.52%, 61.15%, 67.43%, and 61.95% for Ant, ArgoUML, Gantt, and JHotDraw, respectively. Intel-NSGA-II also reduces the



(a) ArgoUML. (b) Ant.

(c) Gantt. (d) JHotDraw.

Fig. 5: Percentage of invalid refactorings across all solutions per generation for NSGA-II, Dep-NSGA-II, and Intel-NSGA-II.



(a) ArgoUML (b) Ant

(c) Gantt (d) JHotDraw

Fig. 6: Percentage of invalid refactorings in refactoring solutions using NSGA-II, Dep-NSGA-II, and Intel-NSGA-II.

percentage of invalid refactorings more quickly than the other algorithms at the population level.

Also, Figure 5 reveals that NSGA-II generates a roughly constant percentage of invalid refactorings equal to or greater than 25%. By introducing the dependency-aware change operators, Dep-NSGA-II reduced the number of invalid refactorings to roughly 15% in the 30th generation. Figure 5 also reveals a major decrease in the number of invalid refactorings

caused by Intel-NSGA-II in the first 12 generations; then it becomes roughly constant and equal to less than 10%. Thus, the number of generations to reach a stable fraction of invalid refactorings is almost the same per algorithm independently from the evaluated project.

Finally, we examined the impact of our proposed change operators at the solution level. Figure 6 shows the distribution of the percentage of invalid refactorings within solutions. Intel-NSGA-II achieves the lowest percentage of invalid refactorings in solutions across all generations for all projects followed by Dep-NSGA-II and NSGA-II, respectively.

Key findings: *Intel-NSGA-II* reduces the percentage of invalid refactorings in the population and refactoring solutions by an average of 65.51% and 43.71% compared to NSGA-II [39] and Dep-NSGA-II, respectively.

TABLE IV: Performance indicators results for NSGA-II, Dep-NSGA-II, and Intel-NSGA-II.

System	Algorithm	I_C	I_{GD}	I_{HV}
ArgoUML	NSGA-II	0.0172	0.0343 ± 0.0342	0.0222 ± 0.0205
	Dep-NSGA-II	0.3172	0.0303 ± 0.0081	0.0349 ± 0.0186
	Intel-NSGA-II	**0.6655**	**0.0262 ± 0.0078**	**0.0801 ± 0.0855**
Ant	NSGA-II	0.0041	0.0242 ± 0.0049	0.0176 ± 0.0205
	Dep-NSGA-II	0.1632	0.0205 ± 0.0047	0.0329 ± 0.0119
	Intel-NSGA-II	**0.8326**	**0.0122 ± 0.0035**	**0.1080 ± 0.0555**
GanttProject	NSGA-II	0.0036	0.0205 ± 0.0027	0.0218 ±0.0111
	Dep-NSGA-II	0.1749	0.0193 ±0.0037	0.0302 ± 0.0209
	Intel-NSGA-II	**0.8215**	**0.0103 ± 0.0024**	**0.1191 ± 0.0536**
JHotDraw	NSGA-II	0.1044	0.0253 ± 0.0050	0.0266 ± 0.0214
	Dep-NSGA-II	0.0413	0.0225 ± 0.0040	0.0349 ± 0.0175
	Intel-NSGA-II	**0.8544**	**0.0136 ± 0.0036**	**0.1341 ± 0.0635**

2) RQ2: Quality: Table IV shows the average I_C, I_{GD}, and I_{HV} of the 30 runs of the three algorithms. The values in bold are the best values achieved for each performance indicator per project. *Intel-NSGA-II* achieved the highest I_{HV} and I_C and the lowest I_{GD} for all projects. Dep-NSGA-II was able to improve the I_{HV}, I_C, I_{GD} compared to NSGA-II by up to 86.93%, 4758.33%, and 15.28%, respectively. Intel-NSGA-II was able to improve the I_{HV}, I_C, I_{GD} compared to NSGA-II by up to 513.63%, 22719.44%, and 49.75%, respectively. This shows that *Intel-NSGA-II* produces better convergence and diversity than the other algorithms.

Table V shows the average quality improvement of solutions, as well as their standard deviations. The bold values are the best values obtained for each metric for each project. *Intel-NSGA-II* produced the best quality improvement in almost all cases. *NSGA-II* produced the lowest quality improvement in 18 out of 24 cases. *Dep-NSGA-II* was able to improve the Effectiveness, Extendibility, Flexibility, Functionality, Reusability, and Understandability compared to *NSGA-II* by an average of 13.31%, 51.89%, 5.61%, 2.07%, 2.28%, 9.54%, respectively. *Intel-NSGA-II* was able to improve the Effectiveness, Extendibility, Flexibility, Functionality, Reusability, and Understandability compared to *NSGA-II* by an average of 17.86%, 46.94%, 83.96%, 64.94%, 57.87%, and 3.54%, respectively. This demonstrates that our intelligent crossover strategy that targets fixing a solution's weaknesses leads to higher quality solutions in the final Pareto-front. There is, however, a performance penalty for the extra work performed by intelligent change operators; on average, execution time doubled. In most cases, this is a more than acceptable trade-off for higher quality refactoring recommendations.

We noticed that NSGA-II never produced the best quality improvement in any cases, which means that the dependency-aware change operators play a significant role in improving the quality of the Pareto-front. In addition, whenever Intel-NSGA-II does not produce "the best quality improvement", the difference between the quality values of Intel-NSGA-II and Dep-NSGA-II is very small. Indeed, the quality improvements rate depends on the number of code smells, size and evolution of the analyzed projects. In our future work, we are planning to validate our approach using more projects to have a clearer understanding of when and why Intel-NSGA-II does not produce "the best quality improvement".

Key findings: *Intel-NSGA-II* outperforms the other algorithms in terms of diversity, convergence, and quality improvement of the Pareto-front using the different evaluation metrics I_C, I_{GD}, and I_{HV} by at least 50% with a modest sacrifice in execution time.

3) RQ3: Relevance: Figure 7 presents the results of manual correctness, precision, and recall for our Intel-NSGA-II algorithm and state of the art refactoring techniques. The detailed responses of the 14 participants can be found in our appendix [7]. Intel-NSGA-II achieved better manual evaluation scores than [39] and existing approaches in all the metrics for all projects. Indeed, the average manual correctness, precision and recall of our algorithm compared to that of Mkaouer et al. [39] are 0.89, 0.82, and 0.87 to 0.67, 0.56, and 0.67 respectively and much better than the remaining tools. Thus, the participants found our refactoring recommendations applicable and consistent with the source code and their design issues. All participants agreed on the benefits of considering dependencies among refactorings when generating refactoring solutions. They mentioned that Intel-NSGA-II increases their trust in refactoring tools and would save them time and effort on filtering out invalid refactorings.

Key findings: *Intel-NSGA-II* provided more relevant and meaningful refactorings than state of the art refactoring recommendation techniques based on manual evaluation of recommended refactorings.

IV. Threats to Validity

Conclusion validity. We used Design of Experiments (DoE) [32] to mitigate the threat related to parameter tuning. DoE is a methodology for systematically applying statistics to experimentation and is one of the most efficient techniques for tuning parameter settings of evolutionary algorithms. Each parameter has been uniformly discretized in intervals. To mitigate the stochastic nature of the search algorithms, we

TABLE V: Average quality improvement of the solutions generated by NSGA-II, Dep-NSGA-II, and Intel-NSGA-II.

System	Algorithm	Effectiveness	Extendibility	Flexibility	Functionality	Reusability	Understandability
ArgoUML	NSGA-II	0.0557 ± 0.0147	0.1484 ± 0.0335	0.0077 ± 0.0077	0.0077 ± 0.0042	0.0130± 0.0051	0.0260± 0.0099
	Dep-NSGA-II	0.0615 ± 0.0112	0.1639 ± 0.0297	**0.0109 ± 0.0084**	0.0082 ± 0.0045	0.0129 ± 0.0054	0.0255 ± 0.0098
	Intel-NSGA-II	**0.0646 ± 0.0174**	**0.1798 ± 0.0332**	0.0094 ± 0.0104	**0.0115 ± 0.0045**	**0.0206 ± 0.0035**	**0.0302 ± 0.0095**
Apache Ant	NSGA-II	0.0177 ± 0.0049	0.0296 ± 0.0112	0.0073 ± 0.0088	0.0070 ± 0.0046	0.0086 ± 0.0021	0.0125 ± 0.0074
	Dep-NSGA-II	0.0214 ± 0.0050	**0.0362 ± 0.0098**	0.0086 ± 0.0085	0.0083 ± 0.0046	0.0099 ± 0.0020	**0.0136 ± 0.0072**
	Intel-NSGA-II	**0.0230 ± 0.0054**	0.0338 ± 0.0098	**0.0164 ± 0.0111**	**0.0123 ± 0.0055**	**0.0139 ± 0.0022**	0.0119 ± 0.0083
GanttProject	NSGA-II	0.0285 ± 0.0077	0.0677 ± 0.0168	0.0045 ± 0.0093	0.0059 ± 0.0048	0.0080 ± 0.0029	0.0098 ± 0.0080
	Dep-NSGA-II	**0.0340 ± 0.0077**	**0.0775 ± 0.0166**	0.0046 ± 0.0107	0.0067 ± 0.0052	0.0094 ± 0.0026	**0.0124 ± 0.0095**
	Intel-NSGA-II	0.0335 ± 0.0097	0.0710 ± 0.0195	**0.0120 ± 0.0153**	**0.0123 ± 0.0073**	**0.0147 ± 0.0033**	0.0093 ± 0.0124
JHotDraw	NSGA-II	0.0451 ± 0.0074	0.1028 ± 0.0175	0.0138 ± 0.0109	0.0122 ± 0.0047	0.0141 ± 0.0028	0.0126 ± 0.0117
	Dep-NSGA-II	0.0463 ± 0.0079	0.1058 ± 0.0191	0.0084 ± 0.0102	0.0085 ± 0.0054	0.0109 ± 0.0040	0.0132 ± 0.0084
	Intel-NSGA-II	**0.0487 ± 0.0111**	**0.1062 ± 0.0193**	**0.0169 ± 0.0176**	**0.0154 ± 0.0084**	**0.0180 ± 0.0041**	**0.0136 ± 0.0137**

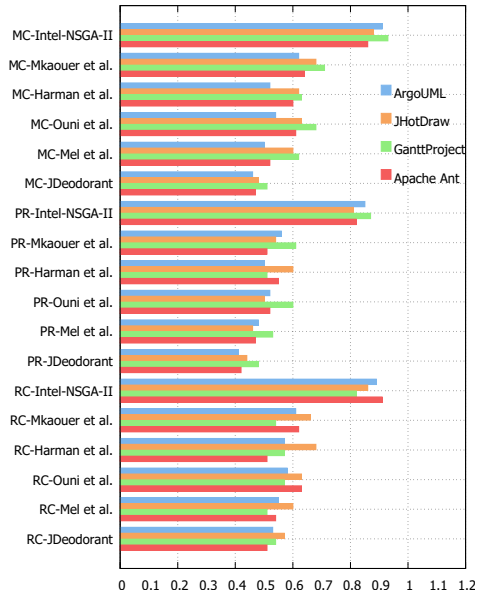


Fig. 7: Manual evaluation of refactoring recommendations generated by the existing multi-objective techniques [23], [52], [48], [39] and the JDeodorant Eclipse plugin [56]).

performed 30 runs per project and algorithm and analyzed the mean results along with the appropriate statistical tests using the Wilcoxon test with a 95% confidence level ($\alpha < 5\%$).

Internal validity. Validation exercise participants had different programming skills and familiarity with refactoring tools. To counter this, we assigned developers to groups according to their experience to reduce the gap between the groups and we adopted a counter-balanced design. Asking the participants to evaluate the refactoring recommendations for all projects would be too much work for them and would reduce the quality of the survey responses. For this reason, we divided the participants into four groups balancing skill level and familiarity with the open-source projects and we asked each one of them to evaluate a single project. We grouped the participants based on their familiarity with the projects to be evaluated. Indeed, it is critical that the participants are knowledgeable about the code of the evaluated projects so they can make accurate judgment about the recommended refactorings. Also, the relatively small number of participants could also be considered a threat to validity. We selected 14 developers to participate in our validation, targeting developers with knowledge of the studied projects. In-depth interviews with a relatively small number of developers familiar with the studied projects yields deep, quality insights that are more useful than those extracted using an online survey with random participants who are not familiar with the studied projects.

Construct validity. Developers might have different opinions about the relevance of recommended refactorings, which may impact our results. Some might think that it is important to refactor, while others might think otherwise. To mitigate this threat, we ensured that each refactoring was evaluated by two developers, and we considered it relevant if both agreed. The overall Cohen's kappa was 0.91 which confirms that there is a significant consensus among developers.

External threats. External threats concern the generalization of our findings. Our validation includes only four projects. One reason for this is to attract more quality responses from survey participants. The more tedious the task that participants must complete, the lower the quality of their responses. The second reason is that running all of the algorithms on all of the projects 30 times takes considerable time.

V. Related Work

A. Search-Based Software Refactoring

Many studies have used search-based techniques to automate software refactoring by optimizing different sets of quality metrics [23], [22], [43], [52], [48], [39], [29], [30], [20], [25], [19], [28], [34], [5], [27], [6]. Kessentini et al. [26] 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 in the source code. Harman and Tratt [23] were the first to use the concept of Pareto optimality in search-based software refactoring to address conflicting quality objectives such as coupling and cohesion. They showed that their multi-objective technique generates better results than a mono-objective approach. Cinnéide et al. [43] also proposed 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. [47] presented a multi-objective refactoring approach to minimize the number of detected defects and maximize the semantic similarity of code elements.

All the above studies used traditional random change operators (e.g. 1-point crossover, random mutation, etc.) that can destroy relevant patterns inside good refactoring solutions when applied randomly, as illustrated in the validation section.

B. Refactoring Dependencies

Chavez et al. [12] investigated how refactoring types affect five quality attributes based on the version history of 23 open source projects. They found that 94% of refactorings are applied to code with at least one low quality attribute value, with 65% of refactorings improving attributes and 35% of all refactorings being neutral on the system. Similarly, Cinnéide et al. [43] studied the impact of individual refactorings on quality attributes, such as using Move Method to reduce the coupling of a class. None of these studies considered the impact of a sequence of refactorings on quality attributes.

Bibiano et al. [11] analyzed batch refactoring characteristics and their effects on code smells in open and closed source projects and concluded that 57% of batches/patterns are simple compositions of only two types of refactorings. They highlight lack of tool support to automatically detect refactoring dependencies as a barrier. However, this study is based on the assumption that refactorings are only related if applied to the same code location, which often is not the case for types of refactorings that modify multiple code fragments. Mens et al. [37] analyzed dependencies at the model-level working with UML. Our work is at the code-level working directly with transformations on the code rather than on UML models where the type of refactorings are different and simplified when compared to the code-level refactorings. Overall, existing studies mainly define what might be better considered similarity relations, such as a collection of refactorings that have similar effects (fixing a code smell) or similar context (applied by the same developer or to the same code location) [36], [59]. None of the existing studies rigorously define refactoring dependencies to integrate them into recommendation tools, including search-based refactoring.

C. Genetic Operators in Search-Based Software Engineering

Search-based software engineering studies proposed few studies on improving the change operators in order to optimize the performance and convergence of search algorithms as well as the quality of generated solutions. However, none of them addressed the refactoring problem or designed new change operators to deal with the issues of solution correctness or the impact of random change operators on solution quality. Oliveira et al. [45] propose a reformulation of program repair operators such that they explicitly traverse three subspaces that underlie the search problem (i.e. Operator, Fault Space, and Fix Space). They implemented new crossover operators that respect the subspace division.

Zhu et al. [61] propose two mechanisms to avoid premature convergence of genetic algorithms: i) dynamic application of crossover and mutation operators; and ii) population partial re-initialization. They implemented two crossover and two mutation operators and, dynamically choose one crossover and one mutation operators to apply in each generation, based on a selection probability that is dependent on average progress. Abido et al. [1] propose improved crossover and mutation algorithms to directly devise feasible offspring chromosomes.

VI. Conclusion

To improve the correctness and quality of refactoring recommendations and increase developer trust in search-based refactoring recommendation tools, we proposed a dependency-aware multi-objective refactoring approach with intelligent change operators that find a balance among quality objectives while reducing the number of invalid refactorings. We evaluated this approach on four open-source projects. We compared our results to existing refactoring techniques that use random change operators, as well as to a dependency-aware technique, to understand the impact of considering refactoring dependencies and fixing quality weaknesses in refactoring solutions. The comparisons show that our proposed approach performs significantly better than the baselines in terms of convergence, diversity, and correctness with a reasonable cost in terms of increased execution time. The survey with 14 practitioners confirmed the relevance of our approach.

For future work, we plan to validate this study with additional programming languages, developers, and projects to generalize our results. We also plan to implement an intelligent mutation operator that targets fixing quality objectives in refactoring solutions. Also, we intend to investigate how many refactoring operations actually depend on each other and how many operations can be executed independently. Another research direction could be to explore further techniques to implement the change operators and compare them with each other. In fact, there are multiple ways of how we choose the refactorings that participate in the mutation and the crossover processes as well as how we perform the change operators. There is a practical balance to study between smarter mutations (more expensive, but more reliable) vs. simpler, more error prone mutations (faster, but not guaranteed). This operation shows benefits from introducing modest constraints in the selection of mutable refactorings. There are certainly other variants that explore different trade-offs between speed and error reduction and that is just for mutating a single refactoring at a time.

Acknowledgements

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. DM21-0720

REFERENCES

[1] M. A. Abido and A. Elazouni. Improved crossover and mutation operators for genetic-algorithm project scheduling. In *2009 IEEE Congress on Evolutionary Computation*, pages 1865–1872. IEEE, 2009.

[2] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976, 2009.

[3] 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*, pages 464–474, 2018.

[4] 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.

[5] 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)*, pages 286–295. IEEE, 2016.

[6] 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)*, pages 1–11. IEEE, 2019.

[7] Anonymous Authors(s). Study appendix, 2021. https://sites.google.com/view/asedependency.

[8] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[9] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto. Recommending refactoring operations in large software systems. In *Recommendation Systems in Software Engineering*, pages 387–419. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[10] G. Bavota, A. D. Lucia, M. D. Penta, R. Oliveto, and F. Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

[11] A. C. Bibiano, E. Fernandes, D. Oliveira, A. Garcia, M. Kalinowski, B. Fonseca, R. Oliveira, A. Oliveira, and D. Cedrim. A quantitative study on characteristics and effect of batch refactoring on code smells. In *Proceedings of 13th the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '19)*, pages 1–11, Porto de Galinhas, Brazil, 2019. IEEE.

[12] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia. How does refactoring affect internal quality attributes? a multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES '17)*, pages 74—-83, Fortaleza, Brazil, 2017. ACM.

[13] 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*, pages 2–es, 2007.

[14] M. C. de Oliveira, D. Freitas, R. Bonifácio, G. Pinto, and D. Lo. Finding needles in a haystack: Leveraging co-change dependencies to recommend refactorings. *Journal of Systems and Software*, 158:110420, 2019.

[15] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[16] 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)*, pages 462–471. IEEE, 2013.

[17] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

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

[19] A. Ghannem, G. El Boussaidi, and M. Kessentini. Model refactoring using examples: a search-based approach. *Journal of Software: Evolution and Process*, 26(7):692–713, 2014.

[20] 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*, pages 175–187, 2011.

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

[22] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2009.

[23] 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*, pages 1106–1113, 2007.

[24] G. Huang, H. Mei, and Q.-x. Wang. Towards software architecture at runtime. *ACM SIGSOFT Software Engineering Notes*, 28(2):8, 2003.

[25] 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*, pages 245–250. Springer, Berlin, Heidelberg, 2013.

[26] 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*, pages 81–90. IEEE, 2011.

[27] 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)*, pages 122–132. IEEE, 2017.

[28] 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*, 97:1–14, 2014.

[29] M. Kessentini, H. Sahraoui, and M. Boukadoum. Example-based model-transformation testing. *Automated Software Engineering*, 18(2):199–224, 2011.

[30] 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*, pages 1–7, 2010.

[31] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoring challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, 2014.

[32] J. Koehler and A. Owen. Computer experiments. handbook of statistics. *Elsevier Science*, pages 261–308, 1996.

[33] M. Kuutila, M. Mäntylä, U. Farooq, and M. Claes. Time pressure in software engineering: A systematic review. *Information and Software Technology*, 121:106257, 2020.

[34] U. Mansoor, M. Kessentini, P. Langer, M. Wimmer, S. Bechikh, and K. Deb. Momm: Multi-objective model merging. *Journal of Systems and Software*, 103:423–439, 2015.

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

[36] H. Melton and E. Tempero. Identifying refactoring opportunities by identifying dependency cycles. In *Proceedings of the 29th Australasian Computer Science Conference (ACSC '06)*, pages 35–41, Australia, 2006. ACM.

[37] T. Mens, G. Taentzer, and O. Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, 2005.

[38] 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)*, volume 1, pages 317–324. IEEE, 2000.

[39] 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*, 21(6):2503–2545, 2016.

[40] 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*, 22(2):894–927, 2017.

[41] M. Mohan and D. Greer. A survey of search-based refactoring for software maintenance. *Journal of Software Engineering Research and Development*, 6(1):3, 2018.

[42] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[43] 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*, pages 49–58, 2012.

[44] M. O'Keeffe and M. O. Cinnéide. A stochastic approach to automated design improvement. In *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java (PPPJ '03)*, pages 59–62, Kilkenny City, Ireland, 2003. ACM.

[45] V. P. L. Oliveira, E. F. Souza, C. Le Goues, and C. G. Camilo-Junior. Improved crossover operators for genetic programming for program repair. In *International Symposium on Search Based Software Engineering*, pages 112–127. Springer, 2016.

[46] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign Champaign, IL, USA, 1992.

[47] 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)*, pages 347–356. IEEE, 2012.

[48] 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)*, 25(3):1–53, 2016.

[49] 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*, 25(3):23, 2016.

[50] 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*, 105:18–39, 2015.

[51] 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*, 83:55–75, 2017.

[52] M. O'Keeffe and M. O. Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, 2008.

[53] 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)*, pages 268–278. IEEE, 2013.

[54] 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*, pages 335–340. IEEE, 2012.

[55] E. Tom, A. Aurum, and R. Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013.

[56] 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*, pages 329–331. IEEE, 2008.

[57] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009.

[58] D. A. Van Veldhuizen and G. B. Lamont. Multiobjective evolutionary algorithm research: A history and analysis. Technical report, Citeseer, 1998.

[59] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. On refactoring support based on code clone dependency relation. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS '05)*, pages 10–pp, Como, Italy, 2005. IEEE.

[60] X. Zhang, Y. Tian, and Y. Jin. A knee point-driven evolutionary algorithm for many-objective optimization. *IEEE Transactions on Evolutionary Computation*, 19(6):761–776, 2014.

[61] F.-l. Zhu, H.-w. Deng, F. Li, and S.-g. Cheng. Improved crossover operators and mutation operators to prevent premature convergence. *Sci Technol Eng*, 10(6):1540–1542, 2010.

[62] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE transactions on Evolutionary Computation*, 3(4):257–271, 1999.