Efficient Refactoring Scheduling Based On Partial Order Reduction

Rodrigo Morales[a,*], Francisco Chicano[b], Foutse Khomh[a], Giuliano Antoniol[a]

[a]*Polytechnique Montréal, Canada*
[b]*Departamento de Lenguajes y Ciencias de la Computación, University of Málaga*

Abstract

Anti-patterns are poor solutions to design problems that make software systems hard to understand and to extend. Components involved in anti-patterns are reported to be consistently related to high changes and faults rates. Developers are advised to perform refactoring to remove anti-patterns, and consequently improve software design quality and reliability. However, since the number of anti-patterns in a system can be very large, the process of manual refactoring can be overwhelming. To assist a software engineer who has to perform this task, we propose a novel approach RePOR (**Re**factoring approach based on **P**artial **O**rder **R**eduction). We perform a case study with five open source systems to assess the performance of RePOR against two well-known metaheuristics (Genetic Algorithm, and Ant Colony Optimization), one conflict-aware refactoring approach and, a new approach based on sampling (Sway). Results show that RePOR can correct a median of 73% of anti-patterns (10% more than existing approaches) with a significant reduction in effort (measured by the number of refactorings applied) ranging from 69% to 85%, and a reduction of execution time ranging between 50% and 87%, in comparison to existing approaches.

Keywords: Software Refactoring, Refactoring Schedule, Anti-patterns, Design Quality, Ant Colony Optimization, Genetic Algorithm

1. Introduction

Refactoring is a software maintenance activity that aims to improve code design, while preserving behavior [1]. In the last decade, many works have reported that refactoring can reduce software complexity, improve developer comprehensibility and also improve memory efficiency and startup time [2, 3]. Hence, developers are advised to perform refactoring operations on a regular basis [4]. However, manual refactoring is a complicated task, as there could be more than one correct solution depending on the design attributes that one is interested in improving. Moreover, the order in which a set of candidate refactorings should be applied is uncertain, and can lead to different designs; some refactorings can have sequential dependencies that require a specific order to enable further refactorings, and other refactorings can be mutually exclusive (i.e., incompatible refactorings). Finding the right sequence of refactorings to apply on a software system is usually a hard task for which no polynomial-time algorithm is known.

To automate the process of anti-pattern resolution through refactoring, some researchers have implemented different metaheuristic techniques [5, 6, 7, 8, 9, 10, 11, 12]. The goal is to find a sequence of refactoring operations that most improves the design quality of a software system. The concept of "quality" here can be interpreted in many different ways. We can reduce the number of design defects, a.k.a., anti-patterns [13] in the software, or improve some desirable quality attributes like maintainability, understandability, design complexity, etc. The problem with existing search-based approaches is that they do not consider how to properly schedule refactorings, but apply metaheuristic techniques blindly, which results in long ex-

*Corresponding author
Email addresses: `rodrigomorales2@acm.org` (Rodrigo Morales), `chicano@lcc.uma.es` (Francisco Chicano), `foutse.khomh@polymtl.ca` (Foutse Khomh), `giuliano.antoniol@polymtl.ca` (Giuliano Antoniol)

ecution times, with certain variations in the results between independent runs, due to the stochastic nature of the techniques applied.

To address the limitations of search-based approaches, some researchers have proposed refactoring approaches that consider conflicts between refactorings. For example, Liu et al. [14] proposed an approach to iteratively select the most promising refactoring operations in terms of design quality, while removing the ones that are in conflict with them, until there are no more refactorings candidates left. In another work, the same research group [15] proposed a refactoring scheme for reducing the effort required for removing different type of anti-patterns using pairwise analysis. The idea is to refactor the anti-patterns that can mitigate the negative effects of other types of anti-patterns, (*e.g.,* removing code duplication also affects anti-patterns related to the size of classes/methods). While these approaches could find a refactoring application order (schedule) to maximize the number of refactorings with the higher quality effect, they do not consider the effort required to apply the proposed sequence on the software system. Moreover, developers are required to provide a list of candidate refactorings as an input and after generating the sequence, they have to apply it manually to the software system.

In this work, we aim to close the gap, by providing automated-refactoring support for developers, that covers all the main steps of the improvement of software design quality through automatic refactoring, i.e., the (1) detection of classes that contain anti-patterns; (2) the generation of refactoring candidates to improve the design quality of the classes detected in (1); (3) the search for an optimal refactoring order; and (4) the application of the refactoring order from (3). To achieve this goal, we propose a new heuristic approach called RePOR (**Re**factoring approach based on **P**artial **O**rder **R**eduction). Partial order reduction is a popular technique for controlling state space explosion in model checking [16]. The intuition is to reduce the number of refactoring sequences to be explored by removing equivalent sequences (*i.e.,* refactoring sequences that leads to the same design). As a result, less search effort is required than when using metaheuristic algorithms. To evaluate RePOR, we conduct a series of experiments over a testbed of five open source software systems (OSS) and compare the results with Genetic Algorithm (GA) [17], Ant Colony optimization (ACO) [18], the conflict-aware refactoring scheduling approach proposed by Liu et al. [14] (referred to as LIU in this paper), and a new optimizer based on sampling (Sway) [19]. We show that the solutions obtained by RePOR overcome the ones obtained by the above-mentioned state-of-the-art optimization techniques in terms of performance (i.e., execution time) and effort (i.e., number of refactorings applied).

Tool and Data Replication. The Eclipse Plug-in and all the data used in the experiments are available on the RePOR replication package [20].

The remainder of the paper is organized as follows: Section 2 discusses the formulation of the refactoring scheduling problem, and describes how to reduce the search-space size using partial order reduction. Section 3 describes RePOR in detail. Section 4 presents the case study for evaluating our approach. Section 5 presents and discusses the results obtained in our case study. Section 7 discloses the threats to the validity of our study. Related work is discussed in Section 8. Finally, we present our conclusions and lay out directions for future work in Section 9.

2. Formulation of the refactoring scheduling problem

As a software system ages, its design quality deteriorates unless it is continually maintained [21]. Refactoring is a software maintenance activity that aims to keep the design quality of a software system at an acceptable level, in order to ensure a normal evolution of the system. Typically, refactoring is performed by applying small transformation operations (*e.g.,* moving a method/field to another class) to a software system while preserving its original behavior. Since there is a wide range of candidate refactorings that can be applied on a system, depending on the domain of the system, an optimal solution may be comprised of several refactorings that improve different quality attributes. Hence, the refactoring scheduling problem consists of finding the best combination of refactorings that maximizes the design quality improvement of a software system. The problem of finding an optimal order can be solved using search-based techniques. Search algorithms start by generating one or more random sequences. Next, the quality of each sequence is computed by applying it to the software system in question, and measuring the improvement in the quality attributes of interest using an objective function (*a.k.a.,*fitness function).

In this work, we evaluate the quality of a refactoring sequence SR as:

$$Q(SR) = \sum_{k \in K} Q(sr_k); \text{ with } Q(sr_k) = AC(k') - AC(k) \quad (1)$$

In Equation 1, SR is a subset of R; R is the set of refactorings to be applied in a system SYS; K is the set of classes in SYS, $K \in SYS$; sr_k is a subset of SR that modifies class k ($k \in K$). Each sub-function $Q(sr_k)$ is computed by subtracting the number of occurrences of anti-patterns in class k after applying sr_k to k (*i.e.,* $AC(k')$) and the number of occurrences of anti-patterns before refactoring (*i.e.,* $AC(k)$). Note that we use the number of occurrences of anti-patterns as a proxy of design quality. The outcome of $Q(SR)$ is a negative value when applying SR to K removes anti-patterns; zero if the number of anti-patterns remains the same, and positive otherwise. The quality effect of applying SR is related to the presence and the order of refactorings in SR.

Hence, we suggest that refactorings should be clustered depending on the classes that they affect. In this way, they can be optimized separately. Since the order of appearance of refactorings that affect different classes in a sequence is irrelevant, we can reduce the number of refactoring operations that we need to evaluate. For example, imagine that we have a set of refactorings: $R = \{A, B\}$ to be scheduled. According to Morales et al. [22], the number of refactoring sequences (S) that we could generate having n refactoring operations is given by Equation 2.

$$S = \begin{cases} \lfloor e \cdot n! \rfloor & \forall n \geq 1 \\ 1 & n = 0 \end{cases} \quad (2)$$

where e is the Euler constant, and n is the number of refactorings available.

Applying Equation 2 to our example gives us 5 possible sequences ($\lfloor e \cdot 2! \rfloor = 5$): $<>, <A>, <B>, <A, B>, <B, A>$, if and only if (iff) we assume that each permutation leads to a different solution (here the term solution refers to the outcome of applying a refactoring sequence to a system, *i.e.,* the resultant design) . Otherwise, $< A, B >$ and $< B, A >$ are two different representations for the same solution and only 4 different solutions exist.

In the case of refactorings that affect the same class, the resultant design may vary depending on the order of application of the refactorings, as the application of one refactoring can enable or disable the rest of refactorings. We can represent the dependency between refactorings as an undirected graph G_B, where an edge $(r_u, r_v) \in E$ exists iff $r_u, r_v \in R_k$. $k \in K$, where K is the set of classes in a system, and R_k is the set of refactorings that affect class k, $R_k \subset R$. G_B, which is a bipartite graph, is linked to the structure of the objective function, where a set of refactorings modify a class, and the application or not of these refactorings affect the number of anti-patterns that remain in this class after refactoring.

We use G_B to find the connected components ($CCAP$). A connected component is a maximal subgraph where all the pairs of vertices are connected by a path. Connected components impose a partial order over the refactoring operations. We borrow the idea of *partial order reduction* from model checking [16], to express the removal of sequences of refactorings that lead to the same design. Partial order reduction (POR) is a method that exploits the commutativity of asynchronous systems to reduce the size of the state space. As concurrent models impose an arbitrary ordering between concurrent events, refactoring scheduling imposes an arbitrary ordering between refactoring operations. The ordering between independent concurrent instructions is meaningless (as the order of independent refactorings is). Hence, we can consider just one ordering for checking one given property since the other orderings are equivalent. This fact can be used to construct a reduced state graph hopefully much easier to explore compared to the full state graph. We leverage this idea to explore only a subset of refactoring permutations that are representative of all refactoring permutations detected.

Another factor that affects the size of the search-space of the refactoring problem is the occurrences of conflicts. We distinguish two kind of conflicts, sequential dependency conflicts and mutual exclusion conflicts. We elaborate more on these two kind of conflicts in the following.

- Given two refactorings r_i and r_j, r_i has a sequential dependency conflict with r_j iff r_j cannot be applied before r_i. We represent sequential dependency conflicts as follows: $r_1 \to r_2$, which means that r_1 can be followed by r_2, but r_2 cannot be followed by r_1. Note that conflicts are directional, i.e., the fact that applying r_j disables r_i does not necessarily means that r_i disables r_j.

- Given two refactorings r_i and r_j, r_i has a mutual exclusion conflict with r_j iff r_i and r_j cannot be applied together in any order. We represent mutual exclusion with the following notation: $r_1 \nleftrightarrow r_2$.

In the extreme case where no conflicts exist among the pairs of refactoring opportunities (i.e., all pairs commute), only the presence or absence of a refactoring opportunity in a sequence makes a difference in the sequence, and the search space can be reduced to 2^n refactoring sequences.

We model the conflicts between refactorings using a directed graph G_C, where the set of refactoring opportunities R is the set of vertices and an edge $e(u, v) \in E$ exists between two refactorings $u, v \in R$ if a conflict exists between u and v. Depending on the refactoring type, we define some heuristics to detect conflicts between refactorings. For example, it is not valid to apply move method to move method m_1 from class A to class B after inlining class A to B, as A will no longer exists, but m_1 now belongs to class B instead, if A is a subclass of B.

To better illustrate the refactoring scheduling problem, and the effect that the consideration of dependencies and conflicts between refactorings has on the size of the search-space, we present an example of the problem in Listing 1.

Listing 1: Example of classes to be refactored

```
class Geometry{
...
 double calcAreaRectangle(Rectangle p1){
         return p1.Width()*p1.Height();
 }
 void longParameterListMethod(int p1, int p2, ..., int
       p15 ){
         ...
 }
}

class Rectangle extends Shape{
     private double width;
     private double height;
     public double Width(){
         return width;
     }
     public double Height(){
         return height;
         }
}

class Shape{
         ...
}
```

The refactorings presented in Table 1 can be applied to refactor the classes described in Listing 1.

Table 1 contains three type of refactorings from [4] that we describe below:

1. Move method. Move a method from one class to another (*e.g.,* to one of its parameter types [23]).
2. Inline Class. If a class contains few responsibilities, move all its features to another class and remove it.
3. Introduce Parameter Object. Replace a list of parameters that typically go together by an object.

Applying Equation 2 to the example shown in Listing 1, we find that the number of refactoring sequences for the code shown in Listing 1 is $S = \lfloor e \cdot 3! \rfloor = \lfloor 16.3097 \rfloor = 16$. A simple manual enumeration, shown in Table 2 confirms this evaluation.

Note that in Equation 2 we assume that a permutation of a subset of refactoring operations always leads to a different software design. However, this assumption may not holds in all cases. In Table 2 we find pairs of refactorings where the application order is irrelevant, *e.g.,* the application order of r_1 and

Table 1: List of refactorings candidates for the example from Listing 1

ID	Type	Source class	Method	Target Class
r_1	Move method	`Geometry`	`calcAreaRectangle`	`Rectangle`
r_2	Inline Class	`Rectangle`	All fields and methods	`Shape`
r_3	Introduce Parameter Object	`Geometry`	`longParameterListMethod`	`GeometryParamObj` *(new)*

Table 2: Enumeration of possible refactoring sequences for the set of refactoring operations {r1, r2, r3}.

sequence	elements	sequence	elements
1.	None	9.	r_3, r_1
2.	r_1	10.	r_3, r_2
3.	r_2	11.	r_1, r_2, r_3
4.	r_3	12.	r_1, r_3, r_2
5.	r_1, r_2	13.	r_2, r_1, r_3
6.	r_1, r_3	14.	r_2, r_3, r_1
7.	r_2, r_1	15.	r_3, r_2, r_1
8.	r_2, r_3	16.	r_3, r_1, r_2

r_3 in sequences 6, 9. Hence, it is possible to reduce even more the search-space by removing these permutations as they lead to the same design (same solution). This occurs because they affect different code segments (the method and target class is different for r_1 and r_3) , *i.e.,* they are unrelated.

In addition, when a conflict exists between refactorings, it is possible to reduce the size of the search space further. For example, consider the sequential dependency conflict between r_1, r_2, that is r_2 cannot be applied before r_1 (inlining class `Rectangle` invalidates any move method refactoring from/to that class). Hence, by removing redundant solutions, and invalid solutions (solutions with elements that are conflicted) we can reduce the search-space size of the motivating example by half (sequences 1, 2, 3, 4, 5, 6, 8 and 11). Thus, the value obtained after applying Equation 2 should be used as an upper bound of the search-space size, as long as we assume that applying a refactoring sequence does not create new refactoring opportunities that were not in the original list. If this happens, the number of possible refactorings can be larger than $|S|$. However, in a typical scenario, software maintainers would repeat the process of finding refactoring opportunities until: (1) it is not possible to apply more refactorings, or (2) they are satisfied with the design quality.

3. Refactoring approach based on Partial Order Reduction

In this section we present RePOR, our novel approach to automate the correction of software anti-patterns through refactoring. RePOR is comprised of 7 steps depicted in Algorithm 1

Algorithm 1: RePOR

```
Input  : System to refactor (SYS), Maximum number of refactoring operations in a
         connected component subgraph (threshold)
Output: An optimal sequence of refactoring operations ($SR$)
1  Require Proc: extractBestPermutation, getFirstValidSequenceFromccap
2  Steps RePOR(SYS, threshold)
3      $AM$=code meta-model generation (SYS)
4      $A$ = Detect Anti-patterns($AM$)
5      $R$ = Generate set of refactoring candidates($AM, A$)
6      $G_B$ = Build Graph of dependencies between refactorings and
         anti-patterns($AM, R, A$)
7      $CCAP$ = Find connected components ($G_B$)
8      $G_C$ = Build Graph of conflicts between refactorings ($AM, LR$)
9      $SR$ = Schedule sequence of refactorings($CCAP, G_C, AM$)
10 Procedure Schedule sequence of refactorings($CCAP, G_C, AM$):
11     $SR = 0$
12     for each ccap $\in CCAP$ do
13         ccap.RemoveInvalidRefactorings($SR$)
14         if ccap.size == 0 then
15             continue
16         else
17             List permuts = enumeratePermutations(ccap)
18             if permuts $\leq$ threshold then
19                 $SR$.addAll(extractBestPermutation($AM, G_C$, permuts))
20             else
21                 $SR$.addAll(getFirstValidSequenceFromccap($AM, G_C, ccap, R$))
22             end if
23         end if
24     end for
25     return $SR$
26 end
```

3.1. Step 1: Code meta-model generation

In this step we generate a light-weight representation (a code meta-model) of a system (SYS), using static code analysis techniques, with the aim of evolving the current design into an improved version in terms of design quality . A code meta-model describes systems at different levels of abstractions. We consider three levels of abstractions to model systems. A code-level model (inspired by UML) which includes all of the constituents found in any object-oriented system: classes, interfaces, methods, and fields. An idiom-level model which is a code-level model extended with binary-class relationships. Examples of binary-class relationships are association, aggregation, and composition relationships with association relationships indicating that one instance of one class "uses" methods and–or fields of the instances of its associated class while composition relationships indicate a constrained association in terms of uniqueness and lifetime. They are identified through static code-analyses with typically 100% precision and recall for associations and a high precision and recall for aggregations. Composition relationships cannot be entirely identified statically because they involve the lifetime of the instances of the classes involved in such relationships. Hence, idiom-level models include association and aggregation relationships and only the few composition relationships that can be identified with high precision and recall statically. A design-level model contains information about occurrences of design motifs, code smells, and anti-patterns. A code meta-model should provide methods to manipulate the design model and generate other models. The objective of this step is to manipulate the design model of a system programmatically. Hence, the code meta-model is used to detect anti-patterns, apply refactoring sequences and evaluate their impact on the design quality of a system. More information related to code meta-models, design motifs and micro-architecture identification can be found in [24, 25].

3.2. Step 2: Detect Anti-patterns

In this step we detect anti-patterns in the meta-model using any available detection tool. The output of this step is a set of anti-patterns instances (A), with the qualified name of the classes and constituents that participate in each detected anti-pattern.

3.3. Step 3: Generate set of refactoring candidates (R)

After we generate a set of anti-patterns that we want to correct from the previous step, we generate a list of refactoring operations based on the type of anti-patterns. For example, in the case of a Blob class, which is a large controller class surrounded by data classes, we may start by moving functionality to related classes in order to reduce size and improve cohesion using *move method* refactoring. We may have more than one possible targets to move a method from the Blob class, which become refactoring candidates, and our approach should be able to select the move method refactoring that improves the most the design quality of the system after refactoring.

3.4. Step 4: Build refactorings dependency graph (G_B)

To avoid evaluating permutations that lead to the same design, it is important to cluster refactorings by the classes they are modifying.

3.5. Step 5: Find connected components ($CCAP$)

To guide the search of refactoring operations, once we have built the refactorings dependency graph (G_B), we proceed to find the connected components of G_B.

3.6. Step 6: Build refactorings conflict graph (G_C)

As we mentioned before, conflicts arise when two or more refactorings affect the same classes or their constituents (fields, methods, etc.). These conflicts should be considered when generating a refactoring schedule to avoid evaluating invalid sequences.

3.7. Step 7: Schedule a sequence of refactorings (SR)

In this final step, we iterate over all connected components, $ccap \in CCAP$, to schedule refactorings that correct more anti-patterns (lines 11-25). At the beginning of the search, the refactorings in sequence SR is empty (line 10). During the search process, we will add refactorings to SR, that can disable other refactorings from R. Hence, we remove refactoring operations that are no longer valid in every iteration of the main loop (line 12). If the number of vertices in a $ccap$ is zero after removing invalid refactorings, we continue with the next connected component. Otherwise, we compute all possible permutations of the refactorings in *ccap* (line 16). To enumerate all permutations of *ccap*, we use *Algorithm G (General permutation generator)* from Knuth [26]. This algorithm generates all permutations with the condition that every permutation is visited only once. Depending on the number of elements in *ccap*, it might not be possible to fit all possible permutations in memory. The input parameter *threshold* is an integer value which represents the maximum number of refactoring operations for a given *ccap* that we can enumerate without decreasing RePOR's performance, and this value is empirically determined according to the architecture of the test computer. If $permuts \leq threshold$, we call *extractBestPermutation* procedure to obtain the best permutation in terms of anti-pattern correction, which is depicted in Algorithm 2. In case the number of permutations is too large to be enumerated (Line 19) we call method *getFirstValidSequenceFromccap* to find the first non-conflicted sequence of anti-patterns from the current *ccap*. We depict the procedure in Algorithm 4.

Algorithm 2 starts by initializing *bestPermutScore* to positive infinity (as we are performing minimization) and *bestPermutation* to an empty list. The main *for-loop* (line 4), consists on iteratively adding refactoring operations from the current permutation to *permut*. If the current refactoring is conflicted with the refactorings already added, it continues to the next operation until the end of the current permutation. Then, it evaluates the impact of the current permutation (*permutScore*), and if this value is less than the current *bestPermutScore*, it replaces *bestPermut* and *bestPermutScore* with *permut* and *bestPermutScore*. Note that the application of each permutation of refactorings can result in one of the following outcomes: the permutation removes an anti-pattern in the source class; it does not remove the anti-pattern in the source class (*e.g.,* there are not enough move method refactorings to substantially decompose a Blob class); removes the anti-pattern in the source class and introduces an anti-pattern in the target class; or does not remove the anti-pattern in the source class, but adds a new

Algorithm 2: Algorithm to extract the best permutation from a list of a set of integers

```
Input  : Code design-model (AM), graph of conflicts $G_C$, list of permutations
         (permuts)
Output: A list of refactorings (bestPermutation)
1  Require Proc: evaluateImpactOfPermutation
2  Procedure extractBestPermutation (AM, $G_C$, permuts):
3      bestPermutScore = $+\infty$
4      bestPermutation = new List
5      for row = 1 to row = permuts.size do
6          permut = new List
7          for col = 1 to col = permuts[1].size do
8              if $G_C$.isTherePathBetweenNodes(permuts[row][col],
                permut)==false then
9                  permut.add(permuts[row][col])
10             end if
11         end for
12         permutScore = evaluateImpactOfPermutation(permut, AM)
13         if permutScore < bestPermutScore then
14             bestPermutation = permut
15             bestPermutScore = permutScore
16         end if
17     end for
18     return bestPermutation
19 end
```

anti-pattern in the target class. The permutation with the best score is returned (line 17).

In Algorithm 3 we present the procedure to evaluate a permutation in terms of the number of anti-patterns that it corrects. The procedure starts by initializing the variable $score = 0$. In line 3, we have a *for loop* to iterate over all refactorings in the permutation. In line 5, we proceed to detect anti-patterns in the vertices adjacents to r in the bipartite graph, *i.e.,* $adj(r, G_B)$. The outcome of the detection is stored in Ap. Next, if the application of r on the code-design model succeeds, we recompute the number of anti-patterns in the related classes, $adj(r, G_B)$ again, this time in the refactored design. Variable *score* is computed by subtracting the count of anti-patterns after refactoring (*i.e.,* Ap') from the count of anti-patterns before refactoring (*i.e.,* Ap) and adding this value to the current *score*. If r cannot be applied to the model, we remove r from *permut* (Line 10). This is done to reduce the overhead of scheduling invalid refactorings. One may think that validating the existence of conflicts between r and the refactorings previously scheduled in Algo-

Algorithm 3: Algorithm to evaluate a permutation in terms of the number of anti-pattern it can remove

```
Input  : A sequence of integers (permut), code-design model (AM), a set of
         refactoring candidates (R), bipartite graph ($G_B$)
Output: An integer value (score)
1  Procedure evaluateImpactOfPermutation(permut, AM, R):
2      score = 0
3      for col = 1 to col = permut.size do
4          r = R.getRefactoring(permut[col])
5          Ap = Detect Antipatterns($adj(r, G_B)$)
6          if AM.ApplyRefactoring(r)==true then
7              Ap' = Detect Antipatterns($adj(r, G_B)$)
8              score=score+(Ap'-Ap)
9          else
10             permut.remove(r)
11         end if
12     end for
13     AM.rollbackSequence(permut,R, $G_B$)
14     return score
15 end
```

rithm 2 should be enough to warrants a valid sequence. However, we cannot be totally sure until we apply the refactoring sequence on the software system. Applying the refactoring on the software system can be computationally expensive, specially for a search algorithm. As an alternative, we use a code-design model that enables us to simulate the application of a refactoring on the software system. At the end of the loop (Line 12), we undo all the refactorings from *permut* that were applied to the code-design model, and return *score* (Lines 13-14).

Algorithm 4 starts at line 2, when variable $desiredEffect$ is set to -1. This means that the application of the sequence built from a *ccap* removes one anti-pattern (in the source class) and do not add any anti-pattern in any related class. Next, a *for loop* (line 5) iterates the elements in *ccap*. If *element* is conflicted with any of the refactorings already scheduled in *sequence*, we skip to the next *element*. Otherwise, we perform anti-patterns detection on the vertices adjacents to r in G_B. The resulting value is stored in Ap. If the application of r succeeds, we retrieve the participating elements of r from the refactored code-design model, and detect anti-patterns again. Next, we add *element* to $tempRefactoringSeq$ and compute *score*, similar to Algorithm 3. If *score* is less or equal to $desiredEffect$, we

Algorithm 4: Algorithm to obtain the first valid sequence from a set of refactorings

```
Input  : Code meta-model (AM), graph of conflicts $G_C$, set of connected
         components (ccap), a set of refactoring candidates (R), bipartite graph ($G_B$)
Output: A sequence of refactorings (sequence)
1  Procedure getFirstValidSequenceFromccap (AM, $G_C$, ccap, R, $G_B$):
2      desiredEffect=-1
3      sequence=new list
4      tempRefactoringSeq=new list
5      for each element ∈ ccap do
6          score=0
7          r = R.getRefactoring(element)
8          if $G_C$.isTherePathBetweenNodes(element, sequence)==true then
9              continue
10         end if
11         Ap = Detect Antipatterns($adj(r, G_B)$)
12         if AM.ApplyRefactoring(r)==true then
13             Ap' = Detect Antipatterns($adj(r, G_B)$)
14             tempRefactoringSeq.add(element)
15             score=score+(Ap' − Ap)
16             if score <= desiredEffect then
17                 removeAntipattern = true
18                 exit for
19             end if
20         end if
21     end for
22     AM.rollback(tempRefactoringSeq)
23     if removeAntipattern = true then
24         sequence = tempRefactoringSeq
25     end if
26     return sequence
27 end
```

set *removeAntipattern* to *true* and exit the main loop. Finally, we rollback the applied refactorings in the code-design model. If we succeeded in removing at least one anti-pattern instance, we set *sequence* equal to $tempRefactoringSeq$. Otherwise, an empty sequence is returned.

4. Case Study

In this section, we conduct a case study to assess the effectiveness of RePOR at improving the design quality of systems. The *quality focus* is the improvement of the design quality of a software system through refactoring. The *perspective* is that of researchers interested in developing automated refactoring tools for software systems, and practitioners interested in improving the design quality of their software systems. The *context* consists of the five metaheuristics: Ant Colony Optimization (ACO), Genetic Algorithm (GA), LIU, Sway, and RePOR, and five open-source systems (OSS). We select Ant Colony Optimization, Genetic Algorithm, LIU to compare the results provided by RePOR as they are well-known techniques successfully used in previous studies for scheduling refactorings [27, 23, 14, 10, 8]. We select Sway because it has been successfully applied to solve a diversity of optimization problems (software product lines, agile project structures, and reducing risk, defects and effort of a project), producing competitive results as those produced by traditional search-based algorithms, but without the need of defining complex transformation operators.

We choose the five OSS according to the following criteria (1): systems belonging to different application domains, (2) availability for replication, (3) use in previous studies concerning refactoring and anti-patterns [28, 10] and (4) non-trivial systems that are likely to present conflict when refactoring.

4.1. Research Questions

We define the following research questions:

(RQ1) To what extent can RePOR remove anti-patterns?
This research question aims to assess the effectiveness of RePOR at improving design quality. We use the number of occurrences of anti-patterns as a proxy for design quality, as they have been found to hinder system evolution [29], and to be correlated with the occurrence of bugs [30]. Hence, the more anti-patterns removed the better.

(RQ2) How does the performance of RePOR compares to the following metaheuristics: ACO, GA, LIU, and Sway, for the correction of anti-patterns?
This research question aims to assess the performance of RePOR in terms of execution time and effort. The rational of studying the execution time is that developers are advise to perform refactoring regularly along with other coding activities [31]. Hence, the waiting time for an algorithm to produce refactoring solutions should be small to be suitable for working on the loop with developers. The rationale for studying effort is that performing a long list of refactorings to achieve high-quality design improvement could lead to an unrecognizable design for developers. It also increases the probability to introduce regression, as it is not suitable to be reviewed by a human pair. Hence, we believe that from developers' perspective [32], it is important to minimize the number of necessary refactorings to attain quality improvement.

4.2. Evaluation Method

In the following, we describe the approach followed to answer **RQ1**, **RQ2**.
All statistics have been performed using the R statistical environment[1]. For all statistical tests, we consider a significance level of 5%. For **RQ1**, we measure the effectiveness of RePOR at removing anti-patterns in software systems using the following *dependent variable*:

- Design Improvement (DI). DI represents the delta of anti-patterns occurrences between the refactored system (SYS') and the original system (SYS) and it is computed using the following formulation.

$$DI(SYS) = \frac{|AC(SYS') - AC(SYS)|}{AC(SYS)} \times 100. \quad (3)$$

[1] http://www.r-project.org/

Where $AC(SYS)$ is the number of anti-patterns in a system SYS and $AC(SYS) \geq 0$. DI, which is a positive real number, represents the improvement amount in percentage, and high positive values are desired. Note that Equation 3 assumes that $AC(SYS') - AC(SYS) < 0$, as RePOR filters out solutions that make the design worse according to the $desiredEffect$ threshold (*cf.,* Algorithm 4).

The independent variable is the refactoring approach applied to each studied system. We statistically compare the number of remaining anti-patterns after refactoring a system using RePOR with the number of remaining anti-patterns when using other refactoring approaches. Specifically, we test the following hypothesis H_{01}: *There is no difference between the number of remaining anti-patterns of a system refactored using RePOR, and a system refactored using other refactoring approaches.* We test the hypothesis using a non-parametric test, i.e., the Mann-Whitney U test [33]. For estimating the magnitude of the differences of means between the number of remaining anti-patterns in systems refactored by RePOR and systems refactored using other approaches, we use the non-parametric effect size measure Cliff's δ (ES), which indicates the degree of overlapping between two sample distributions [34]. ES values range from -1 (if all selected values in the first distribution are larger than the second distribution) to +1 (if all selected values in the first distribution are smaller than the second distribution). It is zero when two sample distributions are identical. Cliff's δ effect size is considered small when $0.147 \leq |ES| < 0.33$, medium for $0.33 \leq |ES| < 0.474$, and large for $|ES| \geq 0.474$ [35].

For **RQ2**, the dependent variables are the execution time and the effort:

- Execution Time (ET). ET represents the total CPU time for the algorithm thread in milliseconds. CPU time is the time that a process is actually running (not waiting on I/O or blocked by other threads that got CPU quantum). We use Oracle's *java.lang.management* library to measure this metric [2].
- Refactoring Effort (RE). We calculate the effort of refactoring by counting the number of refactorings that are scheduled to remove an anti-pattern.

The independent variable is the refactoring approach. We test the following two null hypothesis: H_{02} *: There is no difference between the execution time of RePOR and the execution time of the other studied refactoring approaches.* H_{03} *: There is no difference between the refactoring effort incurred by RePOR and the refactoring effort incurred by other studied refactoring approaches.* To test H_{02}, H_{03}, we use the same statistical tests as in **RQ1**.

4.2.1. Solution representation.

We use a vector representation where each element is a refactoring operation (r) that includes: an *Id* field (unique identifier) to know which refactorings have been applied so far. The anti-pattern's source class, and the type of refactoring. The type of refactoring is used to determine if a conflict with a previous RO in the sequence will arise. In addition to this, we can have more fields providing extra information, *e.g.,* target class and method name for move method, or long method names for Spaghetti code classes.

4.2.2. Code meta-model

The code meta-model is generated using the Patterns and Abstract level Description Language (PADL) model [25]. PADL models are generated by the Ptidej tool suite [36] based on the source code or bytecode of a software system.

4.2.3. Detection and correction of anti-patterns

To detect anti-patterns, we use DECOR as in previous works [8, 7]. DECOR uses a set of rules (metrics, relations

[2] https://docs.oracle.com/javase/8/docs/api/java/lang/management/package-summary.html

between classes) that describe the characteristics of each anti-pattern. In Listing 2 we present an example of the Blob detection rule card. The detection of a Blob is the result of the association of a *mainclass* to one or more *DataClass(es)* (line 2). To detect the main class (i.e., the blob class) the rule used is the result of the union between *LargeClassLowCohesion*, and *ControllerClass* rules (line 3-6). The union operator is interpreted as an addition (a logic OR). *LargeClassLowcohesion* is measured using metrics number of methods plus number of attributes (nmd+nad), and lack of cohesion (LCOM5) using ordinal values (*e.g.,* high, medium, low etc.). These values are computed using the box-plot statistical technique [37] to relate ordinal values with concrete metric values while avoiding setting artificial thresholds. The number after the ordinal value (*i.e.,* VERY_HIGH) represents the degree of fuzziness, which is the acceptable margin around the threshold relative to the ordinal value (line 5,6). The Blob rule card also includes a lexical property, that is the vocabulary used to name the methods and the class (line 8-11), i.e., using words like Process, Control, etc. Finally, it is necessary that the *mainClass* is associated to one or more data class(es). A data class is the one where the *accesor ratio* (number of accessors/number of methods) is greater or equal to 90% (line 12).

Listing 2: Rule card of Blob anti-pattern from DECOR

```
1   RULE_CARD : Blob {
2      RULE : Blob { ASSOC: associated FROM: mainClass
           ONE TO: DataClass MANY };
3      RULE : mainClass { UNION LargeClassLowCohesion
           ControllerClass };
4      RULE : LargeClassLowCohesion { UNION LargeClass
           LowCohesion };
5      RULE : LargeClass { (METRIC: NMD + NAD, VERY_HIGH,
            0) };
6      RULE : LowCohesion { (METRIC: LCOM5, VERY_HIGH,
           20) };
7      RULE : ControllerClass { UNION
8           (SEMANTIC: METHODNAME, {Process, Control,
                Ctrl, Command, Cmd,
9                                    Proc, UI, Manage,
                                          Drive})
10          (SEMANTIC: CLASSNAME,  {Process, Control,
                Ctrl, Command, Cmd,
11                                   Proc, UI, Manage,
                                          Drive,
                                        System,
                                             Subsystem})
                                             };
12     RULE : DataClass { (STRUCT: METHOD_ACCESSOR, 90)
          };
13  };
```

From the set of anti-patterns and code smells detected by DECOR, we consider five types of anti-patterns, namely Blob (BL), Lazy Class (LC), Long Parameter List (LP), Spaghetti Code (SC) and Speculative Generality (SG). These anti-patterns are well-recognized by developers [38], and have been studied in previous works [39, 28, 40, 41].

In Table 3, we present information about the systems studied: number of classes (NOC), number of lines of code $\times 10^3$ (KLOC), and number of anti-patterns detected by type.

Table 3: Descriptive statistics about the studied systems.

System	NOC	KLOC	BL	LC	LP	SC	SG	Total
Apache Ant 1.8.2	697	191	57	40	35	3	6	141
ArgoUML 0.34	1754	183	131	25	281	1	19	457
GanttProject 1.10.2	188	44	47	4	68	5	6	130
JfreeChart 1.0.19	505	98	41	21	62	1	1	126
Xerces 2.7	540	71	56	25	119	2	3	205

The type of refactorings generated to correct the studied anti-patterns are the same that we defined in our previous work [7]. In this work we distinguish between intra-class anti-patterns (anti-patterns in a class, *e.g.,* Long-Parameter List), and inter-class (anti-patterns spreading over more than one class, *e.g.,* Blob).

In Table 4 we describe the type of anti-patterns studied and the refactoring strategies used to remove them. Table 5 shows the number of refactoring candidates that were automatically found in each system.

4.3. RePOR implementation

We instantiate RePOR as an eclipse plug-in and compared it with three refactoring approaches. Design improvement (DI) is measured using Equation 3. To determine the value of the parameter *threshold*, described in Section 3.7, we executed 30 independent executions for each of the systems studied in a Windows 10 64-bit, Intel Core 5 at 2.30 GHz, 12 GB of memory machine, and record the size of *ccap*, where the performance

Table 4: List of studied Anti-patterns and the refactorings used to correct them.

Type	Description	Refactoring(s) strategy
Blob (BL) [13]	A large class that absorbs most of the functionality of the system with very low cohesion between its constituents.	*Move method (MM).* Move the methods that does not seem to fit in the Blob class abstraction to more appropriate classes [23].
Lazy Class (LC) [31]	Small classes with low complexity that do not justify their existence in the system.	*Inline class (IC).* Move the attributes and methods of the LC to another class in the system.
Long Parameter List (LP) [31]	A class with one or more methods having a long list of parameters, specially when two or more methods are sharing a long list of parameters that are semantically connected.	*Introduce parameter object (IPO).* Extract a new class with the long list of parameters and replace the method signature by a reference to the new object created. Then access to this parameters through the parameter object.
Spaghetti Code (SC) [13]	A class without structure that declares long methods without parameters.	*Replace method with method object (RMWO).* Extract long methods into new classes so all local variables become fields on that object.
Speculative Generality (SG) [31]	There is an abstract class created to anticipate further features, but it is only extended by one class adding extra complexity to the design.	*Collapse hierarchy (CH).* Move the attributes and methods of the child class to the parent and remove the *abstract* modifier.

of RePOR is acceptable, and found $threshold = 10$ to be the best trade. The value of *threshold* indicates that for our experiments, we only exhaustively explore the permutations of a *ccap* containing 10 or less refactoring operations, and evaluate the resultant permutations only after removing any conflicted refactoring operation.

The directed graph of conflicts (G_C) is used for the three metaheuristics to avoid scheduling invalid refactorings. Due to the random nature of the metaheuristics studied (i.e., ACO, GA, and SWAY) it is necessary to perform several independent runs to have an idea of the behavior of the algorithms. Hence, we execute 30 independent runs for all the approaches studied and for each system. This is a typical minimum value (i.e., 30 runs) used in the search-based research community to have enough experimental data to perform a statistical analysis.

With respect to the search of the connected components in the graph of dependencies between refactorings (G_B), we use the implementation proposed by Sedgewick and Wayne [42] which uses a recursive depth-first search algorithm.

Table 5: Number of refactoring candidates automatically generated for each studied system.

CH	IC	IPO	MM	RMWO	Total
Ant					
6	9	35	4269	3	4322
ArgoUML					
19	25	281	2475	1	2800
Gantt Project					
6	4	68	3861	5	3944
JfreeChart					
1	21	62	4228	1	4313
Xerces					
3	25	119	4118	2	4267

The stopping criteria for the metaheuristics studied has to be uniform to provide a fair comparison. While in RePOR and LIU the stopping criteria is determined by the number of vertices in the refactoring dependency and conflict graphs, for ACO and GA, the number of evaluations (transformations applied to the randomly-generated initial solutions) required to find an optimal solution cannot be determined before hand. Typically, researchers use number of evaluations or execution time as stopping criteria. We use number of evaluations as the stopping criterion, with a maximum of one thousand evaluations (for each system). This value was empirically determined in our previous works [7, 8].

The next paragraphs disclose in detail the implementations of ACO, GA, LIU, and SWAY used in this case study.

4.4. Ant Colony Optimization Implementation

Ant Colony Optimization (ACO) [18] is a constructive metaheuristic, inspired by the behavior of real ants, that has been successfully applied in solving NP-hard problems, i.e., problems for which no polynomial time algorithm is known, such as routing (traveling salesman, vehicle routing), assignment (graph coloring, frequency assignment), scheduling (job shop, flow shop), network routing (connection-oriented network routing), etc. The benefits of using ACO are: rapid discovery of good solutions, distributed computation which avoid premature convergence like in local search, and greedy heuristics which helps to discover acceptable solutions in the early stages of the search process. In our ACO implementation, the ants are artificial agents that cooperate to build a path in a directed graph $G = (S, T)$ where S is the set of nodes and $T \subseteq S \times S$ is the set of arcs. A finite path over the graph is a sequence of nodes (refactorings operations) $\pi = s_1, s_2, \ldots, s_n$ where $s_i \in S$ for $i = 1, 2, \ldots, n$. We denote π_i the ith node of the sequence and we use $|\pi|$ to refer to the length of the path, *i.e.,* the number of nodes of π.

Our ACO implementation corresponds to a simple ACO [18], where the best ant in the colony updates the pheromone matrix. In Algorithm 5 we describe the main steps of ACO implementation. The steps from Line 2 to 5 are the same steps performed by RePOR, and the main algorithm starts in Line 8. In the algorithm, the path traversed by the ith artificial ant is denoted with a^i. We use $\left|a^i\right|$ to refer to the length of the path, the jth node of the path is denoted with a^i_j, and the last node with a^i_*. We denote with $T(s)$ to the set of successor nodes of node a^i_*. We use the + operator to indicate concatenation between paths. The maximum value for $\left|a^i\right|$ is the number of elements in R (Line 3) *i.e.,* λ_{ant}. The search process starts at Line 8 where the pheromone trails are initialized with the same value: a random number between 0 and 1. After the initialization, the ants start the path construction from different nodes, and the algorithm is executed during a given number of steps m (Line 10). Inside the loop, each ant builds a path randomly selecting the next node according to the pheromone (τ_{ij}) and the heuristic value (η_{ij}) associated to each arc (i, j) (Line 14). In fact, if the *kth* ant is in node i, it selects node j with probability $p_{ij} = \frac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{k \in N_i}[\tau_{ik}]^{\alpha}[\eta_{ik}]^{\beta}}$, where p_{ij} is the probability of an ant to move from node i, to node j. τ_{ij} is the trail intensity which provides information about how many ants have passed through this path. N_i is the set of successor nodes from node i. η_{ij} is an associated heuristic value. k is a mute variable whose domain is the set of successors nodes. The concrete expression is $\eta_{ij} = h(j)$, where $h(j)$ is the score assigned to the candidate refactoring operation by a *heuristic function* (see Section 4.4.1). The construction phase is iterated until the ant reaches the maximum length λ_{ant}, or the current node has no successors in the graph (Line 13).

Once an ant has built a path, it is necessary to evaluate it on-the-fly. We generate a clone of the original design (Line 17) and for each node in a^k we apply its corresponding refactoring operation. Then, the algorithm performs anti-patterns' detection (Line 21) in the resulting model. The design quality is evaluated according to the defined objective function. A good solution is a sequence that corrects more antipatterns.

After the construction phase, the pheromone trails are updated (Line 28) to take into account the quality of the candidate solutions previously built by the ants. The pheromone update follows the expression: $\tau_{ij} \leftarrow \rho\tau_{ij} + f(a^{best}), \forall(i, j) \in a^{best}$, where ρ is the *pheromone evaporation rate* and it holds that $0 \leq \rho \leq 1$. On the other hand, $f(a^{best})$ is the amount of pheromone that the best-ant-path, ever found, deposits on arc (i, j).

The algorithm is finalized whenever the algorithm reaches one of the following conditions:

1. We reach the maximum number of steps (*msteps*).
2. We reach the optimal state, *i.e.,* The number of classes with anti-patterns is zero ($NDC = 0$).

4.4.1. ACO heuristic function

The heuristic value (η_{ij}) is calculated by a function that produces an integer value that defines *how beneficial* is to apply a refactoring r to a class in the system. According to the number of coexisting anti-patterns in the source class, we assign a score that increases with the benefits of applying r on each of the detected anti-patterns in a class. To determine the score, we assign for each refactoring type, an integer value in the range of -2 to 2, where -2 represents a negative effect for a particular anti-pattern, and 2 a very desirable effect, *i.e.,* complete correction. Let us take the following example: suppose that class A has two coexisting anti-patterns namely LC and LP. The suggested refactorings for correcting those anti-patterns are *inline class* and *introduce parameter object*, respectively. Suppose that we want to evaluate the *goodness* of *node* 1, *inline class*. For the first defect (LC) we give a score of **2**, as it is the ideal refactoring for correcting LC, and **0** (no benefit or detriment) to LP; then the total score for *node 1* will be 2 (2+0) as well as for *node 2*. On the contrary, suppose that class *A* has two defects (SC and LP), and we want to prioritize the refactoring of SC over LP. Then, we could assign a heuristic value of 2 to RO type *replace method with method object*, when a class has SC, and 1 to *introduce parameter object*, when a class has LP. In this way the sum of scores for this example will be (2+0), and (1+0) respectively, having more probability to choose the node that corrects SC over the one that corrects LP. The heuristic component η_{ik} cannot accept values equal to zero. Thus, we compute 2^{score} to provide a value in the domain of natural numbers.

In Table 6 we show the parameters used for ACO. These parameters are not set in an arbitrary way, but they are the result of running ACO with different configurations 30 times, in a *factorial design*. For example, to select the importance of the heuristic in ACO, we tried the following couples: no heuristic ($\alpha = 1, \beta = 0$), same importance ($\alpha = 1, \beta = 1$), more importance to pheromone ($\alpha = 2, \beta = 1$) and so on.

Algorithm 5: Ant Colony implementation for scheduling refactoring

```
Input  : System to refactor (SYS)
Output: An optimal sequence of refactoring operations ($SR$)
Steps ACO($SYS$)
    $AM$=code-design model generation (SYS)
    $Ap$ = Detect Anti-patterns($AM$)
    $R$ = Generate set of refactoring candidates($AM, A$)
    $G_C$ = Build Graph of conflicts between refactorings and anti-patterns ($AM, LR$)
    $SR$ = Ant Colony Optimization for refactoring($G_C, AM$)
Procedure Ant Colony Optimization for refactoring($G_C, AM$):
    $\tau = initialize\_pheromone()$
    $step = 1$
    while $step \leq msteps$ AND $Ap \neq 0$ do
        for $k = 1$ to $colsize$ do
            $a^k = null$
            while $\left|a^k\right| \leq \lambda_{ant}$ AND $T(a^k_*) - a^k \neq \emptyset$ do
                $node = select\_successor(G_C, T(a^k_*), \tau, \eta)$
                $a^k = a^k + node$
            end while
            $AM' = AM.clone()$
            for all $node \in a^k$ do
                $apply\_refactorings(AM', node)$
            end for
            $a^k.Ap = detect\_antipatterns(AM')$
            if $DI(a^k) > DI(a^{best})$ then
                $a^{best} = a^k$
                $Ap = a^{best}.Ap$
            end if
        end for
        $\tau = pheromone\_evaporation(\tau, \rho)$
        $\tau = pheromone\_update(\tau, a^{best})$
        $step = step + 1$
    end while
    return $a^{best}$
end
```

Table 6: Parameters of the Ant Colony Optmization algorithm for refactoring scheduling.

Ant Colony Optimization			
Parameter	**Value**	**Parameter**	**Value**
msteps	10	ρ	0.8
colsize	100	β	2.0
λ_{ant}	$\lvert R\rvert$	α	1.0

4.5. Genetic Algorithm implementation

Genetic Algorithm (GA) is an evolutionary metaheuristic [43, 44], where a group of candidate solutions, called individuals or chromosomes, are recombined through some variation operators, *i.e.,* crossover, and mutation, in order to select the best solutions of each iteration (generation). The process of selection and recombination is guided by an evaluation function, *a.k.a.,* fitness function, which ensures that the best individuals have more possibilities to be chosen in each generation. GA is a population-based algorithm, because it works with several solutions at the same time, contrary to trajectory-based methods like hill-climbing and simulated annealing that work with only one solution at a time. The GA used in this work is an elitist genetic algorithm. The proposed elitist GA, selects the best two individuals of the previous population and inserts them directly in the new population (Lines 17 and 18). As next step, the proposed approach considers the best two individuals of the previous population (through a selection operator like binary tournament); cross them over, mutates them, and inserts them in the new population until reaching the stop condition. Our GA implementation [8] is instantiated from a generic one included in JMetal, a Java framework for solving optimization problems [45].

In Algorithm 6, we describe the main steps of our GA implementation. We define P as a list of refactoring sequences (population), and s as a candidate solution (individual) with $s \in P$. Lines from 1 to 6 are the initialization steps and the main algorithm starts in line 7. The population size for the experiments is 100 individuals. In Line 8, the population is initialized with randomly generated refactoring sequences, and evaluated in Line 10. In Line 15, the refactoring sequences are sorted in descending order by their fitness (number of anti-patterns corrected). The main loop starts in Line 16 until the stopping criterion is met. For this case study, we use number of evaluations.

Algorithm 6: Genetic Algorithm implementation for scheduling refactorings

```
   Input  : System to refactor (SYS)
   Output: An optimal sequence of refactoring operations ($SR$)
1  Steps GA($SYS$)
2      $AM$=code-design model generation (SYS)
3      $A$ = Detect Anti-patterns($AM$)
4      $R$ = Generate set of refactoring candidates($AM, A$)
5      $G_C$ = Build Graph of conflicts between refactorings and anti-patterns ($AM, LR$)
6      $SR$ = Genetic Algorithm for refactoring($G_C, AM$)
7  Procedure Genetic Algorithm for refactoring($G_C, AM$):
8      $nPop = populationSize$
9      $P = GenerateInitialPopulation(AM, G_C)$
       /* Evaluation of $P$                                          */
10     for $all\ s \in P$ do
11         $AM' = AM.clone()$
12         $apply\_refactorings(AM', s)$
13         $s.Ap = detect\_antipatterns(AM')$
14     end for
       /* the sequences are sorted in ascendent order according to
          $Ap$                                                       */
15     $P$.sort()
16     while $not\ StoppingCriterion$ do
           /* add the best two individuals of the previous
              population in $O$ population                           */
17         $O.add(P_0)$
18         $O.add(P_1)$
           /* Reproductive cycle                                     */
19         for 0 to $nPop/2 - 1$ do
               /* parents is a list of refactoring sequences         */
20             $parents$ = new List of size 2
21             $parents_0$ = selection_operator($P$)
22             $parents_1$ = selection_operator($P$)
23             $offspring = Variation\_Operators(parents, G_C)$
               /* We generate two offsprings                         */
24             $AM' = AM.clone()$
25             $apply\_refactorings(AM', offspring_0])$
26             $offspring_0.Ap = detect\_antipatterns(AM')$
27             $AM' = AM.clone()$
28             $apply\_refactorings(AM', offspring_1)$
29             $offspring_1.Ap = detect\_antipatterns(AM')$
30             $O.add(offspring)$
31         end for
32         $P$=$O$
33         $O$=null
34         $P$.sort()
35     end while
36     $best\_solution = P_0$
37     return $best\_solution$
38 end
```

4.5.1. Initial Solution Length

The initial length of a solution for new individuals is a random number between one and the total number of refactoring candidates.

4.5.2. Selection operator

The selection operator controls the number of copies of an individual (solution) in the next generations, according to its quality (fitness). Examples of selection operators are tournament selection or fitness proportionate selection [46]. In our implementation we use a binary tournament; the one proposed by Deb et al. [47].

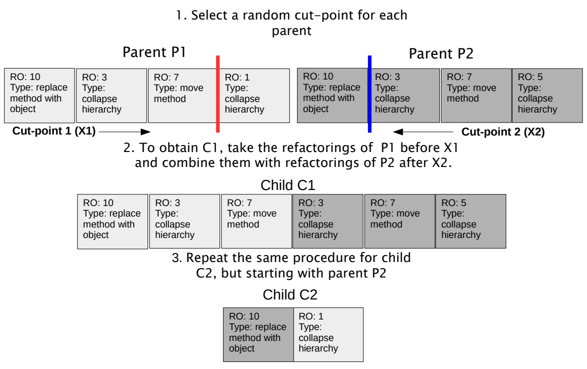
1. Select a random cut-point for each parent
Parent P1
RO: 10 Type: replace method with object
RO: 3 Type: collapse hierarchy
RO: 7 Type: move method
RO: 1 Type: collapse hierarchy
Parent P2
RO: 10 Type: replace method with object
RO: 3 Type: collapse hierarchy
RO: 7 Type: move method
RO: 5 Type: collapse hierarchy
Cut-point 1 (X1)
Cut-point 2 (X2)
2. To obtain C1, take the refactorings of P1 before X1 and combine them with refactorings of P2 after X2.
Child C1
RO: 10 Type: replace method with object
RO: 3 Type: collapse hierarchy
RO: 7 Type: move method
RO: 3 Type: collapse hierarchy
RO: 7 Type: move method
RO: 5 Type: collapse hierarchy
3. Repeat the same procedure for child C2, but starting with parent P2
Child C2
RO: 10 Type: replace method with object
RO: 1 Type: collapse hierarchy

Figure 1: Example of cut and slice technique used as crossover operator.

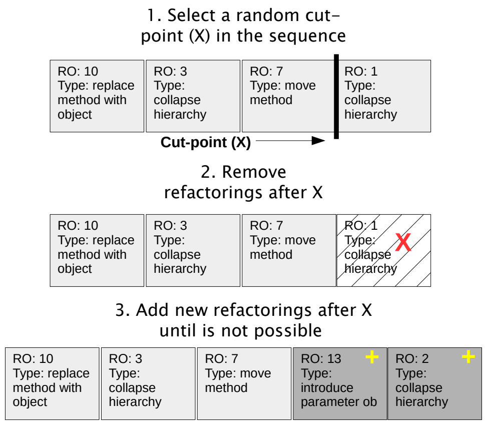
1. Select a random cut-point (X) in the sequence
RO: 10 Type: replace method with object
RO: 3 Type: collapse hierarchy
RO: 7 Type: move method
RO: 1 Type: collapse hierarchy
Cut-point (X)
2. Remove refactorings after X
RO: 10 Type: replace method with object
RO: 3 Type: collapse hierarchy
RO: 7 Type: move method
RO: 1 Type: collapse hierarchy
X
3. Add new refactorings after X until is not possible
RO: 10 Type: replace method with object
RO: 3 Type: collapse hierarchy
RO: 7 Type: move method
RO: 13 Type: introduce parameter ob
RO: 2 Type: collapse hierarchy

Figure 2: Example of the mutation operator used.

4.5.3. *Variation operators*

The variation operators allow metaheuristics to transform a candidate solution so that it can be moved through the decision space in the search of the most attractive solutions, and to escape from local optima. GA uses two main variation operators: crossover and mutation. Crossover consists of combining two or more solutions (known as parents) to obtain one or more new solutions (offspring). We implement the *Cut and splice technique* as crossover operator, which consists in randomly setting a *cut point* for two parents, and recombining with the elements of the second parent's cut point and vice-versa, resulting in two individuals with different lengths. We provide an example in Figure 1.

For mutation, we consider the same operator used in our previous work [8] that consists of choosing a random point in the sequence and removing the refactoring operations from that point to the end. Then, we complete the sequence by adding new random refactorings until there are no more valid refactoring operations to add (*i.e.,* that do not cause conflict with the existent ones in the sequence). We provide an example in Figure 2.

For population size, we use a default value of 100 individuals; and for the probability of applying a variation operator we selected the parameters using a factorial design in the following way: we tested 16 combinations of mutation probability $p_m = (0.2, 0.5, 0.8, 1)$, and crossover probability $p_c = (0.2, 0.5, 0.8, 1)$, and obtained the best results in terms of anti-pattern's correction with the pair $(0.5, 0.8)$.

4.6. *LIU conflict-aware scheduling of refactorings*

Liu et al. [14, 15] proposed different heuristics to solve the refactoring scheduling problem. From these approaches, we select the one in [14], as it is the one that could work with the anti-patterns studied in this paper. On the other hand, the approach proposed in [15] assumes that the refactoring of certain type of anti-patterns can lead to the resolution of another types (*e.g.,* removing code duplications can affect long method). Hence, they leverage this property to remove redundant edges in the graph of conflicts using topological order. However, the type of anti-patterns that we studied and their corresponding refactorings are independent (*e.g.,* it is not appropriate to apply inline class refactoring to a blob Class; collapse hierarchy and inline class cannot be applied at the same time to the same class).

In the following paragraphs we explain the steps that we took to adapt the *conflict-aware scheduling of refactorings* [14] (LIU for short) to our framework, to compare it with RePOR.

LIU uses the QMOOD hierarchical quality model [48] to assess the effect of applying a refactoring on a software system. Because QMOOD combines weighted design metrics (*e.g.,* design size, hierarchies, polymorphism, etc.) to measure quality attributes like reusability, understandability, flexibility, etc. The values obtained for each quality attribute are only useful when compared to the values obtained from systems of the same domain used by the industry. Hence, in the evaluation of LIU [14] they refactored an in-house-developed-modeling tool, and to calibrate the weights of design metrics, they take as an upper-bound the metrics values obtained from a similar open-source project (*BPEL* from Eclipse foundation). However, in this work we use the occurrence of anti-patterns as proxy for design quality. We believe that the occurrences of anti-patterns is a more appropriate way to asses the quality of a software system, as it does not require to find a good-quality representative system to compare with. Our anti-pattern detection framework uses relative values to asses the quality of each class in the system (*cf.,* Section 4.2), which makes it more flexible and easier to adapt for an automated approach as it does not require a calibration step.

The steps of our implementation of LIU are summarized in Algorithm 7.

Algorithm 7: LIU conflict-aware scheduling of refactorings

```
Input  : System to refactor (SYS)
Output: An optimal sequence of refactoring operations ($SR$)
1  Steps LIU($SYS$)
2      $AM$=code-design model generation (SYS)
3      $A$ = Detect Anti-patterns($AM$)
4      $R$ = Generate set of refactoring candidates($AM, A$)
5      $G_C$ = Build Graph of conflicts between refactorings($AM, LR$)
       /* $G_C = (V, E)$                                              */
6      $SR$ = Find sequence of refactorings($G_C$, $AM$)
7  Procedure Find sequence of refactorings($G_C$, $AM$):
8      $SR = \emptyset$
       /* first applying all uninjurious refactorings                 */
9      for each $v_i | adj(v_i) = 0$ do
10         Remove $v_i$ and its edges from $G_C$
11         $SR.add(v_i)$
12     end for
13     if $|G_C| == 0$ then
14         return $SR$
           /* End algorithm                                            */
15     end if
       /* first applying all injurious refactorings                   */
16     for each $v_i | adj(v_i) \neq 0$ do
17         Compute synthetical effect ($SynQ_i$)
18         Compute potential effect ($PQ_i$)
19         Selection and application
20         Update potential effect
21     end for
22     return $SR$
23 end
```

The algorithm starts after generating the list of refactoring candidates and building the graph of conflicts (Lines 2-5). In line 9 we start applying all uninjurious refactorings, *i.e.,* refactorings that do not prevent the application of other refactorings. More formally, i is an uninjurious refactoring iff there is not an edge e from v_i to v_j where $\left\{v_i, v_j \in E\right\}$, $E \in G_C$

If there are no more refactorings left in GC, the algorithm ends (Line 13). Otherwise, we iterate over all injurious refactorings and perform the following steps.

Compute synthetical effect. It consists of computing the effect of applying a refactoring i in the system, *i.e.,* the increment/decrement of anti-patterns occurrences after applying i. We denoted the synthetical effect of applying refactoring i as $SynQ_i$.

Compute potential effect. The application of a refactoring may disable other refactorings (negative effect), or reduce the possibility of conflicts (positive effect) for those refactorings that are adjacent to v_i. Note that for LIU, there is an edge (asymmetrical conflict) between v, u iff u can be applied before, but not after v. In our motivating example, r_2 presents an asymmetrical conflict with r_1 according to LIU. We denoted the potential effect of applying refactoring i as PQ_i.

Selection and application. Select a vertex v_i from G_C that has the greatest potential effect (PQ) and add it to SR.

Update potential effect. Once refactoring i is applied, we remove the vertex from G_C and update the potential effect of vertices adjacents to v_i ($adj(v_i)$).

4.7. Sway *search-based optimizer*

As we briefly discuss in Section 4.5, evolutionary algorithms (EAs) work by improving randomly generated candidate solutions across multiple generations; the evolution process consists of selecting the most prominent individuals to mate and mutate to produce better individuals with certain probability until a stopping condition is reached. To avoid the overhead of transforming candidate solutions, Chen et al. proposed Sway [19], a search-based approach that iteratively clusters candidate solutions from a large unique population to isolate the superior quality solutions. When compared to existing EAs, like NSGAII, SWAY has achieved similar (and sometimes better) results in terms of time and quality of solutions for well-known software engineering problems, but requiring fewer objective evaluations.

We briefly describe the main steps of SWAY.

1. The first step consist of generating a large unique set of solutions.

2. Then, the algorithm splits them in two sets according to the value of their decision variables. Two representative solutions are chosen from each set, called *east* and *west* solutions. Note that SWAY relies on the assumption that there exists a close association between the decision and the objective spaces. If we were to cluster the solutions through their objectives, we would need to evaluate all solutions, which might be computationally expensive.

3. Prune half of the candidates based on the objectives of the representatives of each side (east and west solutions). If none of the representatives of each side *is better* than the other representative call SWAY recursively on each side and join the results.

In the case of the refactoring scheduling problem, a solution is represented by a list of refactoring operations. The objective function is based on the quality improvement achieved after applying a refactoring sequence to a software system.

The steps of our implementation of SWAY are summarized in Algorithm 8.

From Lines 2 to 5, the same initialization steps performed by the other metaheuristics in our framework are found. In Line 9 we set *nPop* equal to the size of population. In this study, we use a population of one thousand individuals. While Chen et al. [19] suggest to use a population of ten thousands or more. The reason why we opt for a lower value is that the execution time was too high in the preliminary experiments we performed with SWAY. However, the size of population that we used in SWAY is ten times bigger than the one we used for ACO and

Algorithm 8: SWAY Algorithm implementation

```
Input  : System to refactor (SYS)
Output: An optimal sequence of refactoring operations ($SR$)
1  Steps SWAY ($SYS$)
2      $AM$=code-design model generation (SYS)
3      $A$ = Detect Anti-patterns($AM$)
4      $R$ = Generate set of refactoring candidates($AM, A$)
5      $G_C$ = Build Graph of conflicts between refactorings($AM, LR$)
6      $SR$ = Find sequence of refactorings($G_C, AM$)
7  Procedure Find sequence of refactorings($G_C, AM$):
8      $SR = \emptyset$
9      $nPop = populationSize$
10     $enough = \sqrt{nPop}$
11     $P = GeneratePopulation(AM, G_C)$
12     $P'$ = SWAY_$F(P, enough)$ /* get the best solution */
13     $bestFitness = 0$
14     for $i = 0$ to $enough - 1$ do
15         $tmpfit = null$
16         if $P'_i.Ap == null$ then
17             $AM' = AM.clone()$
18             $apply\_refactorings(AM', P'_i)$
19             $P'_i.Ap = detect\_antipatterns(AM')$
20             $tmpfit = DI(P'_i)$
21         else
22             $tmpfit = DI(P'_i)$
23         end if
24         if $tmpfit > bestFitness$ then
25             $bestFitness = tmpfit$
26             $SR = P'_i$
27         end if
28     end for
29     return $SR$
30 end
```

GA. We discuss further the execution times we obtained running Sway with different values of *nPop*, and how we reduced the evaluation time of the decision variables in Section 6.

The parameter *enough* (Line 10) is a threshold used as the stop condition for the recursive function Sway $\_F$(*cf.,* Algorithm 9). If the population size is smaller than *enough*, then just return all candidates. Otherwise, Sway $\_F$ splits the candidates in two groups, until this condition is met. The decision spaces in software engineering models have different types of representations, and the one that we use in this work is a sequence of refactoring operations [5, 8]. We define *enough* as in [19], that is, $enough = \sqrt{nPop}$.

In Line 11, P is a set of randomly generated refactoring sequences, of random length. To avoid generating invalid sequences, we skip refactoring operations that conflict with refactorings already inserted in the sequence (according to G_C). In Line 12 we make a call to Sway $\_F$ and retrieve $P' \subset P : |P'| < enough$. It is probable that Sway $\_F$ returns more than one individual. Hence, we extract the best solution from P' (Lines 14-28), and return it as SR (Line 29). Note that we only evaluate the fitness of a solution once (we internally avoid unnecessary fitness evaluations by checking if the count of anti-patterns has been already set) during the execution of Sway. This may happen when comparing the best representatives of each group. Since Sway does not make use of transformation operators, it is safe to assume that the fitness of the individuals remains constant. In Algorithm 9 we present the steps of the recursive procedure Sway $\_F$.

The input of Algorithm 9 is a list of refactoring sequences, and the stop condition (set in Algorithm 8). The output is a pruned list of the best sequences derived from recursive calls of Sway. Sway works by splitting the individuals in two groups according to their decision variables. The way to evaluate the decision variables depends on the representation chosen. Then Sway prunes half of them based on the fitness value of the representative individuals, and not the whole population. The SPLIT procedure is responsible of finding the representative of each group (east, west). If the west solution is not better than the east solution, then the set *eastItems* is explored using Sway recursively, to obtain a set of solution that is stored in Δ_1. The same procedure, replacing the east solution by the west solution and *eastItems* by *westItems* is done to obtain set Δ_2. The SPLIT function has to be designed according to the solution representation, in our case it is a refactoring sequence type. In the original publication of Sway [19], there is no implementation of the SPLIT function for this representation. Hence, we designed a SPLIT procedure based on the suggestions of the authors of Sway.

Algorithm 9: Sway recursive algorithm

```
   Input  : refactoring sequences (items), stop condition (enough)
   Output: pruned results
1  Require Proc: SPLIT, BETTER
2  Procedure Sway _F (items):
3      if $items.size < enough$ then
4          return items
5      else
6          $\Delta_1, \Delta_2 = \emptyset, \emptyset$
7          $west, east, westItems, eastItems = SPLIT(items)$
8          if $\neg BETTER(west, east)$ then
9              $\Delta_1 = $ Sway_F$(eastItems)$
10         end if
11         if $\neg BETTER(east, west)$ then
12             $\Delta_2 = $ Sway_F$(westItems)$
13         end if
14         return $\Delta_1 \cup \Delta_2$
15     end if
16 end
```

In Algorithm 10 we present the SPLIT procedure used in our implementation of Sway. Algorithm 10 starts by initializing Obj_δ, east and west representatives (e, w) and the two groups of individuals $(westItems, eastItems)$. Next a random refactoring sequence r is selected from *items*, and $furthestPermutation(r, items)$ computes the furthest permutation from any element in *items* to r, called e. That is $max(D)$ where $D = (d_0, d_1..d_n), \forall d = DISTANCE(r, item) : item \in Items$ (*cf.,* Algorithm 11). To find w, we compute $furthestPermutation(e, items)$. Then, we compute the fitness' difference between the two representatives (Obj_δ). In line 19, a loop to find the two representatives with the largest Obj_δ based on a predefined threshold (*attempts*) is executed. We arbitrary set the value of *attempts* to 10 as suggested by the authors of Sway, while any value from 1 to items divided by 2 would be acceptable. Note that the longer the value of attempts is, the higher the execution time.

Algorithm 10: SPLIT algorithm for refactoring sequence type

```
   Input  : refactoring sequences (items), stop condition (enough)
   Output: representatives (west, east), two groups (westItems, eastItems)
1  Require Proc: DISTANCE,
2  Procedure SPLIT (items,enough):
3      $Obj_\delta = 0, e = \emptyset, w = \emptyset, westItems = \emptyset, eastItems = \emptyset$
4      $r = getRandomItem()$
5      $e = furthestPermutation(r, items)$
6      $w = furthestPermutation(e, items)$
7      if $e.Ap == null$ then
8          $AM' = AM.clone()$
9          $apply\_refactorings(AM', e)$
10         $e.Ap = detect\_antipatterns(AM')$
11     end if
12     if $w.Ap == null$ then
13         $AM' = AM.clone()$
14         $apply\_refactorings(AM', w)$
15         $w.Ap = detect\_antipatterns(AM')$
16     end if
17     $Obj_\delta = abs(DI(e) - DI(w))$
18     $count = 1$
19     repeat
20         $r = getRandomItem()$
21         $tempe = furthestPermutation(r, items)$
22         $tempw = furthestPermutation(e, items)$
23         $tempObj_\delta = abs(DI(tempe) - DI(tempw.Ap))$
24         if $tempObj_\delta > Obj_\delta$ then
25             $e = tempe, W = tempw$
26         end if
27         $count = count + 1$
28     until count=attempts
29     for each item $\in$ items do
30         if $DISTANCE(item, e) < DISTANCE(item, w)$ then
31             $eastItems.add(item)$
32         else
33             $westItems.add(item)$
34         end if
35     end for
36     return $e, w, westItems, eastItems$
37 end
```

The final step (line 29), consists of mapping large amounts of refactorings sequences to their corresponding group ($eastItems$, $westItems$).

In the refactoring scheduling problem, a refactoring sequence is a permutation derived from a subset of a list of refactoring candidates, that have to be applied in a specific order, and where the occurrence or absence of any operation from the complete list of refactoring candidates determines the final quality of the refactoring sequence.

Selecting the best metric to measure the distance between two sequences is not straightforward, and it has been found to be a NP-hard problem [49], which means that most probably only approximation algorithms can be efficient for large $n : n = |permutation|$. Deterministic alternatives, includes Kendall Tau Distance (K) [50]. We suggest that an adequate distance metric is the one that considers the features of each solution according to the problem to solve.

To measure the distance of two refactoring sequences, K distance requires that the two permutations being of equal size. This cannot be granted, since the existence of conflict between refactoring operations prevents this to happen. Alternatively, we could artificially alter one of the permutations, like adding the missing elements from the largest permutation to the end of the other, to measure the distance using K, but this adds overhead to control for it and does not represent the nature of our problem. For that reason, we propose a metric that considers (1) the presence or absence of elements in two permutations; (2) the order of appearance of the elements that both permutations have in common.

In Algorithm 11, we present our proposed procedure to compute the distance between two refactoring sequences.

Algorithm 11 starts by setting *aux* to the set difference between $RS1$ and $RS2$; next, *distance* is assigned the value of *aux* length. Similarly, we subtract $RS2$ and $RS1$ into *aux*, and

Algorithm 11: Distance algorithm for refactoring sequence type

```
Input  : Refactoring sequences ($RS1, RS2$)
Output: Distance ($distance$)
Procedure DISTANCE ($RS1, RS2$):
    $distance = |RS1 \setminus RS2| + |RS2 \setminus RS1|$
    $nbcomElem = |RS1 \cap RS2|$
    for $i = 1$ to $i \leq |nbcomElem|$ do
        for $j = i + 1$ to $j \leq |nbcomElem|$ do
            if $RS1_i < RS1_j$ then
                if $RS2_i > RS2_j$ then
                    $distance = distance + 1$
                end if
            else
                if $RS2_i < RS2_j$ then
                    $distance = distance + 1$
                end if
            end if
        end for
    end for
    return $distance$
end
```

adding *aux* length to distance. At this point *distance* represents the elements that are absent in both permutations. The next step, is a pairwise comparison to obtain the relative position in which the operations in common appear (Lines 8-21), and in case the relative position changes in a pair the distance is incremented by one. After making the pairwise comparison, between the common operations, the final distance is returned (line 23).

Finally, for determining $BETTER$ from two permutations (p_1, p_2), let $d = DI(p_1) - DI(p_2)$, if $d > \epsilon$, p_2 is better. Otherwise, p_1 is better. Where $\epsilon = 0 : DI(p_1) - DI(p_2) = \epsilon \; iff \; DI(p_1) = DI(p_2)$.

5. Results

In this section, we answer our two research questions that aim to evaluate RePOR.

RQ1: To what extent can RePOR remove anti-patterns?

We present in Table 7 the Design improvement (DI) in general and for different anti-pattern types, for each studied system. The results are the median of the 30 independent executions.

We observe that overall, the design improvement (first column) of the solutions generated by RePOR is higher in comparison with the improvements achieved by the other approaches. The DI of LIU is close to the one obtained by RePOR except in one system, JfreeChart, where ACO and GA performed better than LIU. Concerning ACO and GA, the DI achieved is very close between them, but lower than the one achieved by RePOR. Sway is even inferior to the one achieved by the rest of the approaches by 10% approximately.

With respect to the type of anti-patterns, RePOR have some difficulty to remove Blob anti-patterns compared to the other metaheuristics (we discuss further in Section 6), with one exception, Ant, where it improves more than LIU. In the case of Ant, and Gantt systems, RePOR achieved the same design improvement than Sway. For Lazy Class, the results achieved by RePOR are the same compared to ACO, GA and LIU except for Gantt, where it removes less instances than the others. Sway attained less improvement for 4 out 5 systems. The different was Gantt where Sway surpass RePOR.

For Long-Parameter List, RePOR attained the best results in all the systems studied. For Spaghetti code, RePOR overcomes the rest of the approaches in Gantt, and tie with ACO, GA, and LIU on the other systems. Finally, for Speculative Generality no difference exists between the five approaches, except in Ant, and ArgoUML where the results of Sway were inferior.

Table 8 presents the Mann-Whitney test results and Cliff's δ effect size (ES) obtained when comparing the number of remaining anti-patterns of the systems after being refactored by RePOR and the other refactoring approaches. We observe that all the differences are statistically significant with a large effect size, except for JFreeChart where the difference between ACO and RePOR is small, and the pair GA-RePOR where the effect size is negligible. Therefore we reject H_{01} for the rest of the systems.

Table 7: Design Improvement (%) in general and for different anti-pattern types.

Metaheuristic	DI	DI_{BL}	DI_{LC}	DI_{LP}	DI_{SC}	DI_{SG}
Ant						
ACO	57.45	68.42	22.5	74.29	66.67	100
GA	58.16	68.42	22.5	74.29	66.67	100
LIU	58.87	54.39	22.5	100	66.67	100
RePOR	60.28	57.89	22.5	100	66.67	100
Sway	45.36	57.89	20	60	66.67	83.33
ArgoUML						
ACO	75.93	51.15	100	83.63	100	100
GA	76.59	51.15	100	84.7	100	100
LIU	81.40	50.38	100	92.88	100	100
RePOR	81.62	38.93	100	98.58	100	100
Sway	62.91	48.09	84	66.01	100	86.84
Gantt Project						
ACO	60	17.02	100	83.82	70	100
GA	60.77	14.89	100	85.29	80	100
LIU	63.85	14.89	100	92.65	60	100
RePOR	66.15	8.51	75	100	100	100
Sway	50	8.51	100	70.59	60	100
JfreeChart						
ACO	75.4	39.02	100	89.52	100	100
GA	75.4	39.02	100	90.32	100	100
LIU	72.22	31.71	100	88.71	100	100
RePOR	75.4	24.39	100	100	100	100
Sway	61.90	36.59	90.48	73.39	100	100
Xerces						
ACO	56.59	14.29	100	65.55	100	100
GA	57.56	14.29	100	67.23	100	100
LIU	64.39	16.07	100	78.99	50	100
RePOR	73.17	5.36	100	98.32	100	100
Sway	41.87	14.29	68.00	49.58	50	100

Table 8: Pair-wise Mann-Whitney U Test test for design improvement.

Pair	$p-value$	Cliff's δ	Magnitude
Ant			
ACO-RePOR	2.561349e-12	1	Large
GA-RePOR	1.431438e-11	1	Large
LIU-RePOR	1.685298e-14	1	Large
Sway-RePOR	1.190193e-12	1	Large
ArgoUML			
ACO-RePOR	1.176641e-12	1	Large
GA-RePOR	1.143381e-12	1	Large
LIU-RePOR	1.685298e-14	1	Large
Sway-RePOR	1.206843e-12	1	Large
Gantt Project			
ACO-RePOR	1.036681e-12	1	Large
GA-RePOR	1.086586e-12	1	Large
LIU-RePOR	1.685298e-14	1	Large
Sway-RePOR	1.165138e-12	1	Large
JfreeChart			
ACO-RePOR	0.06868602	0.2333333	Small
GA-RePOR	0.2771456	-0.1333333	Negligible
LIU-RePOR	1.685298e-14	1	Large
Sway-RePOR	1.183399e-12	1	Large
Xerces			
ACO-RePOR	1.0618e-12	1	Large
GA-RePOR	9.946555e-13	1	Large
LIU-RePOR	1.685298e-14	1	Large
Sway-RePOR	1.193116e-12	1	Large

We reject the null hypothesis H_{01} for Ant, ArgoUML, Gantt, JfreeChart, and Xerces. In these five systems, the number of remaining anti-patterns after refactoring using RePOR is significantly lower than the number of anti-patterns remaining in the systems after refactoring using the other refactoring approaches (i.e., ACO, GA, LIU and Sway*). With respect to the magnitude of Cliff's δ, the difference is large for all the systems, except the pairs ACO-RePOR and GA-RePOR in JFreeChart, where it is small and negligible, respectively. Overall, our results suggest that for the set of anti-patterns studied and the systems analyzed, RePOR can correct more anti-patterns, than ACO, GA, LIU and* Sway.

RQ2: How does the performance of RePOR compares to the following metaheuristics: ACO, GA, LIU, and* Sway*, for the correction of anti-patterns?

We present in Table 9 the execution time (ET) and the effort (EF) incurred for each refactoring scheme. ET is given in seconds, while EF represents the number of refactorings applied. The results are the median of 30 independent runs.

Table 9: Median performance metrics for each system, metaheuristic. Execution time (ET) is in seconds, and the effort (EF) is the number of refactorings applied.

Metaheuristic	Execution Time	Effort
Ant		
ACO	11505.73	1686.00
GA	11558.97	1676.00
LIU	260.45	1641.00
RePOR	82.05	827.00
Sway	7301.64	1624.00
ArgoUML		
ACO	5617.51	1119.00
GA	5664.39	1123.00
LIU	148.45	1166.00
RePOR	72.88	438.00
Sway	2833.33	1020.00
Gantt Project		
ACO	5924.93	1069.00
GA	5975.71	1067.00
LIU	652.45	894.00
RePOR	133.45	119.00
Sway	1779.60	981.00
JfreeChart		
ACO	11321.81	1748.00
GA	11369.82	1748.00
LIU	877.74	1747.00
RePOR	133.30	297.00
Sway	13677.25	1654.00
Xerces		
ACO	5781.67	886.00
GA	5831.93	887.00
LIU	389.43	909.00
RePOR	63.07	178.00
Sway	1777.49	819.00

We can observe that RePOR performs better than the other algorithms in terms of execution time and effort, with a remarkable difference, while removing more anti-patterns and using less resources. In terms of execution time; it takes between one minute and less than three minutes to generate a sequence for a complete system, while the second best scheme (LIU) takes a median of six and half minutes. The number of refactorings scheduled are considerably less than the other approaches, which will likely make RePOR attractive to software developers and maintainers, specially when applying refactoring while performing other maintenance tasks.

The performance of GA and ACO is poor compared to RePOR, despite using the same solution representation and the conflict graph (to discard invalid refactorings). We attribute this poor performance to their incapability to discard equivalent sequences (*i.e.,* permutations of refactorings that lead to the same design). Despite the fact that LIU has integrated a mechanism to evaluate the potential effect of applying/removing a refactoring from a sequence, it cannot avoid scheduling uninjurious refactorings that do not improve the design quality, incurring additional costs in effort and time. Sway, as we expected, perform faster than ACO and GA even that the population size is ten times larger. However, the execution time spent by Sway is considerably high compared to the ones of LIU and RePOR. The same pattern occurs with respect to the effort (number of refactorings scheduled). Here we observe that scheduling less refactorings for Sway affect the design improvement achieved, while in RePOR scheduling less refactorings did not affect the design improvement attained.

Concerning RePOR, the overhead occurs when generating a refactoring sequence from a permutation, in case that it contains a large number of elements. To deal with this issue, RePOR only considers a subset of refactoring operations from the permutation until it reaches the *desiredimpact*, *i.e.,* the correction of an anti-pattern instance without introducing a new one (*cf.,* Algorithm 4). However, we do not expect to find many cases where the number of elements in a connected component is to large to be exhaustively explored. In Table 10 we provide some statistics about the size of the connected components in G_B generated by RePOR from the studied systems. We can observe that the median size of the connected components is one, and the number of connected components with size greater than one goes from 11% to 47% of total number of connected components in the worse scenario.

Table 10: Statistics of the connected components (*CCAP*) in G_B from the studied systems

System	Median size	Size>1	Total *CCAP*
Ant	1	46	99
ArgoUML	1	46	424
Gantt Project	1	25	108
Jfreechart	1	30	106
Xerces	1	36	173

ACO, GA, and Sway are algorithms for which it is not possible to predict when an optimal solution will be found. In general, the performance of a metaheuristic can be affected by the correct selection of its parameters. The configurable settings of the search-based techniques used in this paper correspond to stopping criterion, population size, and the probability of the variation operators (except for Sway). We use the number of evaluations as the stopping criteria for ACO and GA, while Sway relies on the parameter *enough* for that purpose.

In the case of ACO and GA, as the maximum number of evaluations increase, we expect the algorithm to obtain better quality results. The increase in quality is usually very fast when the maximum number of evaluations is low. That is, the slope of the curve quality versus maximum number of evaluations is high at the very beginning of the search. But this slope tends to decrease as the search progresses. Our criterion to decide on the maximum number of evaluations is to select a value for which this slope is low enough. In our case *low enough* is when we observe that no more anti-patterns are removed after *n* number of evaluations, where *n* is the value that we are testing. We empirically tried different values in the range of 100 to 1500 and found 1000 to be the best value. However, that does not imply that the best solution is to be found at the end of the 1000 iterations, but could happen before. In addition, computing the average of design improvement with respect to time could help to determine if the evolution trend of the solutions could reach its inflexion point, or the algorithm was stopped prematurely.

To study the evolution of the quality of the solutions obtained by each algorithm every time the current best solution is improved, we compute the average quality of each solution with respect to time, and present the results in Figure 3. The quality is expressed as DI, and the time is normalized using the min-max normalization, that is the minimum time value is mapped to 0 and the maximum value to 1. Given that RePOR and LIU produce only one solution in the entire process (instead of producing several solutions and evolving them), there is only one point for these approaches. Note that we exclude Sway from this analysis, since its goal is show that evolving an initial population is not necessary, but performing fast sampling is enough to filter the best individuals.

In Figure 3, the interpretation for a point p (t,v) is: from t and until the next sample, the average quality of the metaheuristic is v, where t represents time and v is the DI. We can observe that RePOR produces high-quality solutions in a small fraction of time, in comparison to the other approaches. There are only two cases where differences are small: in ArgoUML, LIU is very close to the results achieved by RePOR in terms of quality with a difference of 0.2%, and incurring only 1.75% additional time, while the difference with the best solutions of ACO and GA is not less than 5%. In JfreeChart, where GA approaches the best solution found by RePOR with a difference of 0.02% in DI, but with a remarkable difference of 99.70% of additional time. This is the only case where GA and ACO are clearly better than LIU. For the rest of the systems, as it is shown in Figure 3, both metaheuristics reached their inflexion point far below the optimal solutions found by RePOR and LIU.

With respect to the type of refactorings applied, we present in Table 11 the number of refactorings applied by type. We can observe that the number of refactorings applied by RePOR are similar to those applied by the other metaheuristics, except for move method. That explains the reduction in effort required by RePOR compared to the other metaheuristics. It also explains why the results obtained for the removal of Blob are not so good, since for this type of anti-pattern requires the application of many refactorings to be corrected. Still, this should not be considered as a flaw of our approach, since the main objective is to correct the largest number of anti-patterns without prioritizing the correction of a particular type of anti-pattern, over the others anti-patterns. In this regard, RePOR succeeds well in improving the design quality of the systems studied, in a reasonable amount of time.

Table 11: Median count of refactorings applied for each system, refactoring scheme, by type.

Metaheuristic	Collapse Hierarchy	Inline Class	Introduce Param-Obj.	Move Method	Replace Method with Obj.
Ant					
ACO	6	9	256	1643	3
GA	6	9	27	1629	3
LIU	6	9	35	1589	2
RePOR	6	9	35	774	3
Sway	5	8	23	1584	2
ArgoUML					
ACO	17	24	246	829.5	1
GA	18	23	249	828.5	1
LIU	18	23	281	843	1
RePOR	17	25	280	115	1
Sway	15	20	198	783	1
Gantt Project					
ACO	6	4	59	996	3
GA	6	4	60	994	3
LIU	6	4	68	812	4
RePOR	6	4	68	37	5
Sway	6	4	52	916	3
JfreeChart					
ACO	1	21	56	1669	1
GA	1	21	56	1669	1
LIU	1	21	62	1662	1
RePOR	1	21	62	212	1
Sway	1	19	49	1588	1
Xerces					
ACO	3	25	97.5	758.5	2
GA	3	25	99	759	1.5
LIU	3	25	119	761	1
RePOR	3	25	119	29	2
Sway	3	17	79	715	1

Finally, to assess the statistical significance of the results obtained, we compare performance metrics between RePOR and each metaheuristic using the same procedure as **RQ1**. Table 12 presents the pair-wise statistical tests for each metaheuristic. We observe that all the differences are statistically significant with a large effect size. Therefore we reject H_{02} for the five studied systems.

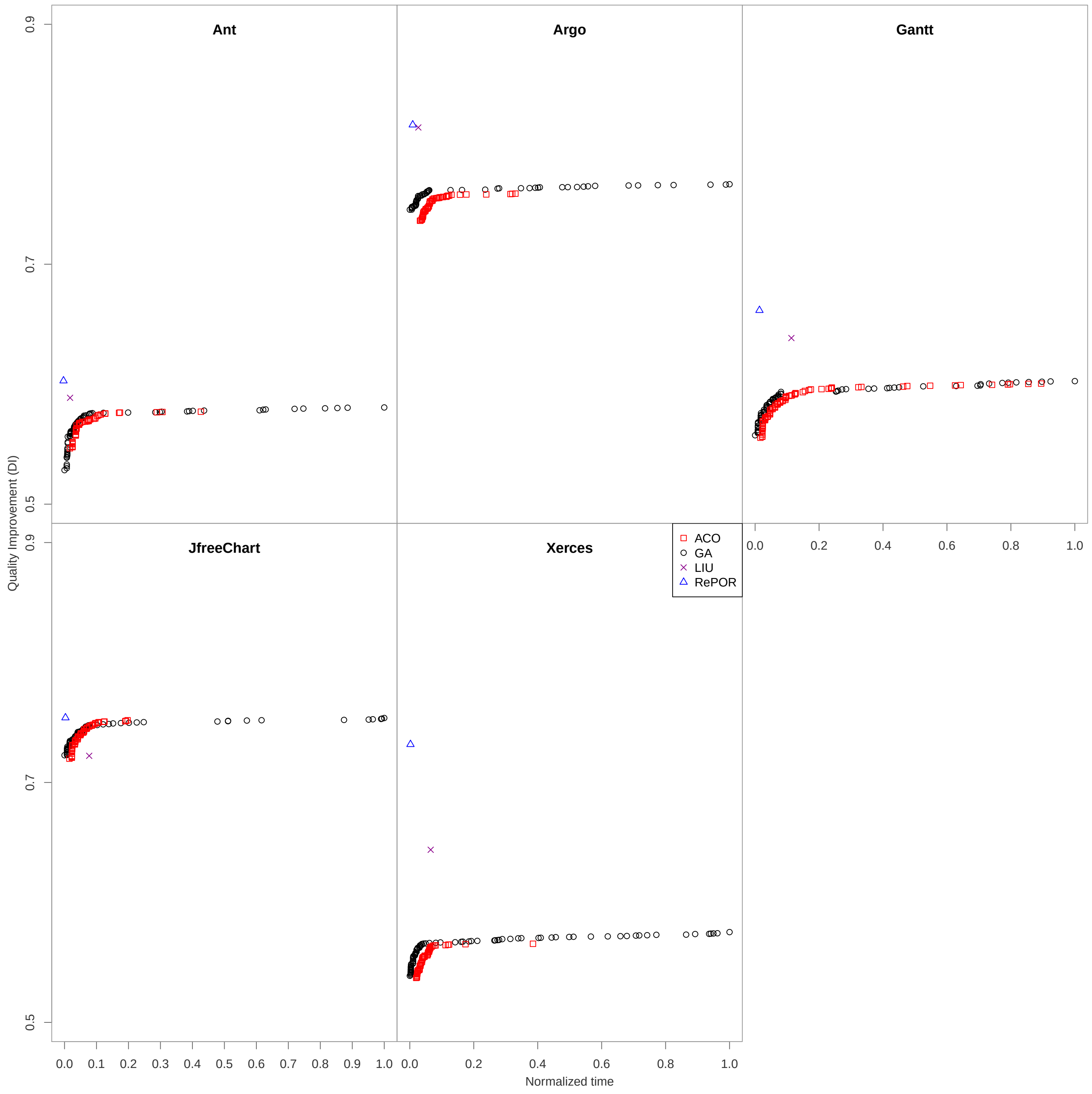

Figure 3: Quality evolution of the refactoring solutions with respect to time.

Table 12: Pair-wise Mann-Whitney U Test test for performance metrics.

Metric	Pair	$p-value$	Cliff's δ	Magnitude
Ant				
ET	ACO-RePOR	1.691123e-17	1	Large
EF	ACO-RePOR	1.133109e-12	1	Large
ET	GA-RePOR	1.691123e-17	1	Large
EF	GA-RePOR	1.197023e-12	1	Large
ET	LIU-RePOR	1.691123e-17	1	Large
EF	LIU-RePOR	1.685298e-14	1	Large
ET	SWAY-RePOR	1.691123e-17	1	Large
EF	SWAY-RePOR	1.209803e-12	1	Large
ArgoUML				
ET	ACO-RePOR	1.691123e-17	1	Large
EF	ACO-RePOR	1.191166e-12	1	Large
ET	GA-RePOR	1.691123e-17	1	Large
EF	GA-RePOR	1.202906e-12	1	Large
ET	LIU-RePOR	1.691123e-17	1	Large
EF	LIU-RePOR	1.685298e-14	1	Large
ET	SWAY-RePOR	1.691123e-17	1	Large
EF	SWAY-RePOR	1.209803e-12	1	Large
Gantt Project				
ET	ACO-RePOR	1.691123e-17	1	Large
EF	ACO-RePOR	9.750474e-13	1	Large
ET	GA-RePOR	3.017967e-11	1	Large
EF	GA-RePOR	1.13497e-12	1	Large
ET	LIU-RePOR	1.691123e-17	1	Large
EF	LIU-RePOR	1.685298e-14	1	Large
ET	SWAY-RePOR	1.691123e-17	1	Large
EF	SWAY-RePOR	1.209803e-12	1	Large
JfreeChart				
ET	ACO-RePOR	1.691123e-17	1	Large
EF	ACO-RePOR	1.038395e-12	1	Large
ET	GA-RePOR	1.691123e-17	1	Large
EF	GA-RePOR	1.124768e-12	1	Large
ET	LIU-RePOR	1.691123e-17	1	Large
EF	LIU-RePOR	1.685298e-14	1	Large
ET	SWAY-RePOR	1.691123e-17	1	Large
EF	SWAY-RePOR	1.209803e-12	1	Large
Xerces				
ET	ACO-RePOR	1.691123e-17	1	Large
EF	ACO-RePOR	1.144319e-12	1	Large
ET	GA-RePOR	1.691123e-17	1	Large
EF	GA-RePOR	1.175678e-12	1	Large
ET	LIU-RePOR	1.691123e-17	1	Large
EF	LIU-RePOR	1.685298e-14	1	Large
ET	SWAY-RePOR	1.691123e-17	1	Large
EF	SWAY-RePOR	1.200942e-12	1	Large

We reject the null hypothesis H_{02} and H_{03}, for Ant, ArgoUML, Gantt, JfreeChart, and Xerces. In these five systems, the execution time and the effort incurred by RePOR are significantly lower than those incurred by the other refactoring approaches. With respect to the magnitude of Cliff's δ, the difference is large for all the systems analyzed. Overall, our results suggest that for the set of anti-patterns studied and the systems analyzed, RePOR can correct more anti-patterns, using less time, and requiring less effort (in terms of refactorings applied) than ACO, GA, LIU and SWAY.

6. Discussion

In this section we discuss the results obtained by RePOR and their relevance for software maintainers and toolsmiths interested in improving the design quality of a software system through refactoring.

In Section 5 we have shown that RePOR is able to correct more anti-patterns using considerably less resources in terms of time and effort than state-of-art refactoring approaches. However, we observed that the number of instances of Blob anti-pattern removed by RePOR was lower than the number of Blobs removed by the other approaches. This could be explained by the large amount of refactorings that are required to remove a Blob anti-pattern, in comparison to other types of anti-patterns. Another interesting observation is the fact that Long Parameter List and Lazy class anti-patterns show higher improvement with RePOR. Therefore, there seems to be a trade off between the refactorings that can be scheduled, as it is not possible to improve all types of anti-patterns to the same extent. What we present in this paper is an alternative refactoring approach, which proves to be more efficient than existing refactoring approaches in terms of design improvement, execution time, and effort. We achieved this result by clustering refactorings by the class that they affect in a connected component subgraph (*ccap*), and exhaustively searching (when possible) the best order for the refactorings for each *ccap*, as they are likely to lead to a different software design. In addition, as each *ccap* may contain conflicted refactorings that cannot be scheduled simultaneously, these refactoring operations are removed from the search space too, reducing the length of the sequences to be evaluated. Finally, for the set of refactorings in a *ccap* where the size is too large to explore all permutations exhaustively, we implement in our approach a mechanism to stop the addition of refactorings if we found that the desired effect (*i.e.,* the desired improvement in quality) is achieved, or just simply when the permutation does not lead to any improvement (*i.e.,* does not correct any anti-pattern). In comparison, LIU approach runs until there is no more refactorings left in the graph, so it assumes that all the refactorings that are not conflicted have to be scheduled. An assumption that may lead to the inclusion of unnecessary refactorings in the final sequence. With respect to ACO and GA, they start with random initial solutions that are iteratively transformed until the stopping criteria is achieved. While this proved to be useful for removing Blob anti-patterns, the usage of resources in terms of time and effort seems to be prohibitive for a coding session or when working interactively with a developer, and may be more suitable for refactoring sessions running after-hours as a batch process. Another disadvantage of ACO and GA is that they have to be calibrated in order to perform reasonably well, with the plethora of parameters involved for each algorithm as we show in Section 4. With respect to Sway, we observe that beside the number of evaluations were reduced, the cost of evaluating the decision variables using a refactoring sequence representation was expensive, that is the $DISTANCE$ metric. To address this problem, we store the results of the $DISTANCE$ in a map structure and considering that the result is commutative (*i.e.,* the $DISTANCE(i, j)$ is the same that $DISTANCE(j, i)$), we managed to reduce the execution time considerably. For example, in ArgoUML we managed to reduce the execution time from 5 hours to just 1. It is probable that if we increase the size of the population, we could achieved better results, but the cost would be counterproductive. For example, in our preliminary experiments, we set the size of the population for ArgoUML to $10,000$, and measure an execution time of more than 12 hours. Yet, the design improvement reached 70.24%, 10% higher than with the value reported in this paper. But still inferior compared to the other metaheuristics employed in this study. The main weakness that we see in using Sway for the refactoring schedule problem is that the population is randomly generated (similar to GA and ACO), so it is difficult to reach a good solution, but as the authors mentioned [19], Sway is a good alternative algorithm for benchmarking other metaheuristics.

One final remark, the refactoring sequences generated by all the approaches studied, do not prioritize any code entities that a developer might be interested. It is possible that developers are interested in refactoring certain packages or classes from which they have the ownership; or simply they just avoid to touch legacy code or critical components. To provide developers with a tool that could be used during daily coding tasks, we integrate RePOR as an Eclipse plug-in [20]. After analyzing a software system (or a subset of classes), our plug-in presents information about the anti-patterns detected, and a generates a refactoring sequence where they can select the refactorings that they consider appropriate.

7. Threats to validity

We now discuss the threats to validity of our study following common guidelines for empirical studies [51].

Construct validity threats concern the relation between theory and observation. Our case study assumes that each anti-pattern is of equal importance, when in reality, this may not be the case. Concerning the scheduling of refactorings, we assume that the potential refactoring operations that can be applied in a software system are determined before the refactoring process begins. This is a big assumption, as new refactoring operations might be found as a consequence of changes in the code, *e.g.,* the application of previous refactorings. However, the search for new refactoring opportunities after applying each refactoring in a sequence is a costly operation. Therefore, most (if not all) the works on automatic refactoring assume that there is a list of refactoring opportunities at the beginning of the search and the optimization algorithm simply selects which of them will be applied and their order until the end of the list/starting of a new refactoring session [22].

Threats to internal validity concern our selection of subject systems, tools, and analysis method. With respect to anti-pattern's detection, DECOR is known to be accurate [28], it is not possible to guarantee that we detect all anti-patterns or that what we detect as anti-patterns are indeed true anti-pattern instances. Other anti-pattern detection techniques and tools should be used to confirm our findings.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the constructed statistical models. In particular, we used non-parametric tests that do not require any assumption on the underlying probability distribution of data.

Reliability validity threats concern the possibility of replicating this study. Every result obtained through empirical studies is threatened by potential bias from data sets [52]. To mitigate these threats we tested our hypotheses over five open-source systems with different size, purpose and years of development. In addition to this, we attempt to provide all the necessary details required to replicate our study. The source code repositories of Apache Ant, ArgoUML, JfreeChart, Gantt and Xerces are publicly available, and have been studied in previous studies related to anti-patterns and code smells. In addition, we made the tool and the data generated publicly-available through our on-line replication web site [20].

Threats to external validity concern the possibility to generalize our results. Our study is focused on five open source software systems having different sizes and belonging to different domains. Nevertheless, further validation on a larger set of software systems is desirable, considering systems from different domains, as well as several systems from the same domain. In this study, we used a particular yet representative subset of anti-patterns as proxy for software design quality. Future works using different type of anti-patterns are desirable.

8. Related Work

The work related to RePOR can be divided in two categories: refactoring scheduling, and search-based refactoring.

8.1. Refactoring Scheduling

In this category we present some of the representative works that proposed techniques to schedule rationally a set of candidate refactorings to improve the effect of refactoring according to their defined quality objectives.

Mens et al. [53] formulated a model to analyze refactoring dependencies using critical pair analysis. However, this model lacks of automation to schedule refactorings once potential conflicts are detected.

Bouktif et al. [54] proposed an approach to schedule refactoring actions in order to remove duplicated code using genetic algorithm.

Liu et al. [14] proposed an heuristic algorithm to schedule refactorings based on a conflict matrix and the effects of candidate refactorings on the design. They evaluated their approach on a house-made modeling tool using QMOOD [48] model and found that it outperforms a manual approach.

Liu et al. [15] proposed an algorithm to schedule the refactoring of code smells using pairwise analysis. By using topological sort on graph that represents the type of anti-patterns detected, they reduced the search of sequences by removing redundant edges that correspond to overlapping smells. However, they did not automate the application of the refactorings on the systems.

These previous works require a list of candidate refactorings in advance to schedule unlike our approach, which automatically detects anti-patterns and propose refactoring candidates.

Lee et al. [55] proposed an approach to automatically schedule refactorings to remove method clones using a Competent Genetic Algorithm. The proposed approach was evaluated using a testbed of four open-source systems. They found higher quality improvement compared to manual and greedy search in terms of QMOOD model [48], but the same quality improvement using exhaustive search for the less complex systems. Zibran and Roy [56] proposed an approach to refactor code clones based on constraint programming. They evaluated their approach on four in-house systems and reported that it outperformed greedy and manual approaches.

Moghadam and Ó Cinnéide [57] proposed an approach for refactoring scheduling where the quality goal is set to be a desired design expressed as a UML model, and the refactoring operators are transformations aimed at achieving that model. They evaluated their approach on an open-source system with a small set of 50 refactorings to be scheduled and found that the produced sequence of refactorings could transform the initial design into the desired design with 100% of success.

8.2. Search-based refactoring

Seng et al. [23] proposed an approach based on GA, that aims to improve the cohesion of the entities through the implementation of *move method* refactoring. They evaluated the quality of the refactoring sequences with a weighted sum fitness function that comprise coupling, cohesion, complexity and stability measurements. O'Keeffe and Cinnéide [58] proposed an approach that relies on the QMOOD model [48] to assess the quality of the candidate refactorings. They tested their approach using local (SA and hill climbing) as well as global search techniques (GA). Although they proposed hill climbing as the most suitable technique for search-based refactoring, they did not find statistically significant difference among the other techniques with respect to quality gain.

Harman and Tratt [5] introduced a multi-objective refactoring approach for improving two compromised metrics in software design: coupling, and the standard-deviation of number of methods per class. They showed that using the concept of Pareto optimality, it is is possible to find the *Pareto front*, which is the set of solutions where there is no component that can be improved without decreasing the quality of another component. Thus, the outcome is a not a single but a set of optimal solutions to be selected by the developer.

Concerning *swarm* optimization, Fawad and Heckel [59] formulated the refactoring scheduling problem as a graph transformation problem, using ACO. However, they did not empirically assessed the performance of their approach, or compared it to the performance of other metaheuristics. Simmons et al. proposed an interactive ACO algorithm to support software designers in the early stages of the software development process (ELSD) [60]. The idea is to interactively find the best candidate design by grouping relevant methods and attributes into classes and present them to the designer to provide feedback. They conclude that ACO is effective in finding useful solutions faster than other multiobjective metaheuristics. In our approach the perspective is the code that already exists and needs to be maintained.

Moghadam and Ó Cinnéide [6] proposed an automated approach where the goal is to reach a desired design, described as an UML diagram through the mapping of the model differences (between the UML diagram of the source code and the desired model) into source level refactorings. The difference with our approach, is that the software designer needs to provide in advance a desired design, to allow the approach to generate the refactoring sequences required to achieved this desired design. Something that is not always feasible.

Ouni et al. [11] proposed a multi-objective evolutionary algorithm approach based on the NSGA-II [47]. The two conflicting objectives of this approach are correcting a large quantity of design defects, while preserving semantic coherence.

Mkaouer et al. [61] proposed an interactive refactoring approach that allow users to rank candidate solutions found by a NSGA-II algorithm.

Recently, researchers have been adding new objectives and refining existing pitfalls in the existing implementations of NSGA-II for refactoring [62, 7, 63].

Our approach differs from these works in the following points: (1) while most of the recent works implemented EAs, our approach reduces the search space by implementing techniques derived from partial order reduction which leads to faster results and less effort; (2) current approaches require the user to input a set of defect examples to generate the detection rules, however, in practice the availability and quality of such datasets compromise the recall and precision of the correct identification of anti-patterns.

9. Conclusion

In this paper, we presented RePOR, a novel approach for automatically scheduling refactoring operations for correcting anti-patterns in software systems. To evaluate RePOR, we conducted a case study with five open-source software systems and compared the performance of RePOR with the performance of two well-known metaheuristics (GA and ACO), one conflicting-aware refactoring approach (LIU), and a recent metaheuristic based on sampling (SWAY). Results show that RePOR can correct more anti-patterns than the aforementioned techniques in just a fraction of the time, and with less effort. In the future we plan to extend the evaluation of RePOR, considering more open and close source software systems, and more quality attributes, *e.g.,* energy efficiency.

10. Acknowledgments

We would like to thank Jianfeng Chen and Tim Menzies for their support and suggestions during the implementation of SWAY for this work.

11. References

[1] W. F. Opdyke, Refactoring object-oriented frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign (1992).

[2] B. D. Bois, S. Demeyer, J. Verelst, T. Mens, M. Temmerman, Does god class decomposition affect comprehensibility?, in: IASTED Conf. on Software Engineering, 2006, pp. 346–355.

[3] B. van Rompaey, B. Du Bois, S. Demeyer, J. Pleunis, R. Putman, K. Meijfroidt, J. C. Dueas, B. Garcia, Serious: Software evolution, refactoring, improvement of operational and usable systems, in: Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conf. On, 2009, pp. 277–280.

[4] M. Fowler, Refactoring: improving the design of existing code, Pearson Education India, 1999.

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

[6] I. H. Moghadam, M. O. Cinneide, Code-imp: A tool for automated search-based refactoring, in: Proceedings of the 4th Workshop on Refactoring Tools, IEEE Computer Society, 2011, pp. 41–44.

[7] R. Morales, A. Sabane, P. Musavi, F. Khomh, F. Chicano, G. Antoniol, Finding the best compromise between design quality and testing effort during refactoring, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1, 2016, pp. 24–35.

[8] R. Morales, Z. Soh, F. Khomh, G. Antoniol, F. Chicano, On the use of developers context for automatic refactoring of software anti-patterns, Journal of Systems and Software 128 (2017) 236 – 251.

[9] M. O'Keeffe, M. O. Cinneide, Search-based software maintenance, in: Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on, 2006, pp. 10 pp.–260.

[10] A. Ouni, M. Kessentini, H. Sahraoui, M. Boukadoum, Maintainability defects detection and correction: a multi-objective approach, Automated Software Engineering 20 (1) (2013) 47–79.

[11] A. Ouni, M. Kessentini, H. Sahraoui, M. S. Hamdi, Search-based refactoring: Towards semantics preservation, in: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, IEEE, 2012, pp. 347–356.

[12] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, M. S. Hamdi, Improving multi-objective code-smells correction using development history, Journal of Systems and Software 105 (0) (2015) 18 – 39.

[13] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, T. J. Mowbray, Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, 1*st* Edition, John Wiley and Sons, 1998.

[14] H. Liu, G. Li, Z. Y. Ma, W. Z. Shao, Conflict-aware schedule of software refactorings, IET Software 2 (5) (2008) 446.

[15] H. Liu, Z. Ma, W. Shao, Z. Niu, Schedule of bad smell detection and resolution: A new way to save effort, IEEE Transactions on Software Engineering 38 (1) (2012) 220–235.

[16] A. Lluch-Lafuente, S. Edelkamp, S. Leue, Partial order reduction in directed model checking, in: International SPIN Workshop on Model Checking of Software, Springer, 2002, pp. 112–127.

[17] J. H. Holland, Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence., U Michigan Press, 1975.

[18] M. Dorigo, V. Maniezzo, A. Colorni, Ant system: optimization by a colony of cooperating agents, Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on 26 (1) (2006) 29–41.

[19] J. Chen, V. Nair, R. Krishna, T. Menzies, "sampling" as a baseline optimizer for search-based software engineering, IEEE Transactions on Software Engineering (2018) 1–1doi:10.1109/TSE.2018.2790925.

[20] R. Morales, F. Chicano, F. Khomh, G. Antoniol, RePOR replication package (2017).
URL "http://www.swat.polymtl.ca/rmorales/jssrepor/"

[21] D. L. Parnas, Software aging, in: ICSE '94: Proc. of the 16th Int'l conference on Software engineering, IEEE Computer Society Press, 1994, pp.

279–287.

[22] R. Morales, F. Chicano, F. Khomh, G. Antoniol, Exact search-space size for the refactoring scheduling problem, Automated Software Engineering (2017) 1 – 6.

[23] O. Seng, J. Stammel, D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, GECCO 2006: Genetic and Evolutionary Computation Conference, Vol 1 and 2 (2006) 1909–1916.

[24] Y.-G. Gueheneuc, H. Albin-Amiot, Recovering binary class relationships: Putting icing on the uml cake, ACM SIGPLAN Notices 39 (10) (2004) 301–314.

[25] Y.-G. Guéhéneuc, G. Antoniol, Demima: A multi-layered framework for design pattern identification, Software Engineering, IEEE Transactions on 34 (35) (2008) 667–684.

[26] D. Knuth, The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, no. pt. 1 in algorithms, Pearson Education, 2014.

[27] F. Qayum, R. Heckel, Local search-based refactoring as graph transformation, in: Search Based Software Engineering, 2009 1st International Symposium on, IEEE, 2009, pp. 43–46.

[28] N. Moha, Y.-G. Gueheneuc, L. Duchien, A. Le Meur, Decor: A method for the specification and detection of code and design smells, Software Engineering, IEEE Transactions on 36 (1) (2010) 20–36.

[29] M. Abbes, F. Khomh, Y.-G. Gueheneuc, G. Antoniol, An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, in: Software Maintenance and Reengineering (CSMR), 2011 15th European Conf. on, 2011, pp. 181–190.

[30] F. Khomh, M. D. Penta, Y.-G. Gueheneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical Softw. Engg. 17 (3) (2012) 243–275.

[31] M. Fowler, Refactoring – Improving the Design of Existing Code, 1^{st} Edition, Addison-Wesley, 1999.

[32] E. Murphy-Hill, A. P. Black, Refactoring tools: Fitness for purpose, Software, IEEE 25 (5) (2008) 38–44.

[33] M. Hollander, D. A. Wolfe, E. Chicken, Nonparametric statistical methods, John Wiley & Sons, 2013.

[34] N. Cliff, Ordinal methods for behavioral data analysis, Psychology Press, 2014.

[35] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, L. Devine, Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohensd indices the most appropriate choices, in: annual meeting of the Southern Association for Institutional Research, 2006, pp. 1–51.

[36] Y.-G. Guéhéneuc, Ptidej: Promoting patterns with patterns, in: Proceedings of the 1st ECOOP workshop on Building a System using Patterns. Springer-Verlag, 2005, pp. 1–9.

[37] J. M. Chambers, Graphical Methods for Data Analysis, 1st Edition, Wadsworth International Group, 1983.

[38] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, Do they really smell bad? a study on developers' perception of bad code smells, in: Software Maintenance and Evolution (ICSME), 2014 IEEE Int'l Conference on, IEEE, 2014, pp. 101–110.

[39] F. Khomh, M. D. Penta, Y.-G. Gueheneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical Softw. Engg. 17 (3) (2012) 243–275.

[40] E. Van Emden, L. Moonen, Java quality assurance by detecting code smells, in: Reverse Engineering, 2002. Proceedings. Ninth Working Conference on, IEEE, 2002, pp. 97–106.

[41] F. Khomh, S. Vaucher, Y.-G. Gueheneuc, H. Sahraoui, A bayesian approach for the detection of code and design smells, in: Quality Software, 2009. QSIC'09. 9th International Conference on, IEEE, 2009, pp. 305–314.

[42] R. Sedgewick, K. Wayne, Algorithms, 4th Edition, Addison-Wesley, 2011.

[43] D. Whitley, A genetic algorithm tutorial, Statistics and computing 4 (2) (1994) 65–85.

[44] D. Whitley, An overview of evolutionary algorithms: practical issues and common pitfalls, Information and software technology 43 (14) (2001) 817–831.

[45] J. J. Durillo, A. J. Nebro, jmetal: A java framework for multi-objective optimization, Advances in Engineering Software 42 (2011) 760–771.

[46] B. L. Miller, D. E. Goldberg, Genetic algorithms, tournament selection, and the effects of noise, Complex systems 9 (3) (1995) 193–212.

[47] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: Nsga-ii, Evolutionary Computation, IEEE Transactions on 6 (2) (2002) 182–197.

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

[49] L. Bulteau, G. Fertin, I. Rusu, Sorting by transpositions is difficult, SIAM Journal on Discrete Mathematics 26 (3) (2012) 1148–1180. arXiv:https://doi.org/10.1137/110851390, doi:10.1137/110851390.
URL `https://doi.org/10.1137/110851390`

[50] M. G. Kendall, A new measure of rank correlation, Biometrika 30 (1/2) (1938) 81–93.
URL `http://www.jstor.org/stable/2332226`

[51] R. K. Yin, Case Study Research: Design and Methods - Third Edition, 3rd Edition, SAGE Publications, 2002.

[52] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, Software Engineering, IEEE Transactions on 33 (1) (2007) 2–13.

[53] T. Mens, G. Taentzer, O. Runge, Analysing refactoring dependencies using graph transformation, Software and Systems Modeling 6 (3) (2007) 269–285.

[54] S. Bouktif, G. Antoniol, E. Merlo, M. Neteler, A novel approach to optimize clone refactoring activity, in: Proceedings of the 8th annual conference on Genetic and evolutionary computation, ACM, 2006, pp. 1885–

1892.

[55] S. Lee, G. Bae, H. S. Chae, D. Bae, Y. R. Kwon, Automated scheduling for clonebased refactoring using a competent ga, Software: Practice and Experience 41 (5) (2011) 521–550.

[56] M. F. Zibran, C. K. Roy, A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring, in: 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation, 2011, pp. 105–114.

[57] I. H. Moghadam, M. O. Cinnide, Resolving conflict and dependency in refactoring to a desired design, e-Informatica Software Engineering Journal 9 (1).

[58] M. O'Keeffe, M. O. Cinneide, Getting the most from search-based refactoring, Gecco 2007: Genetic and Evolutionary Computation Conference, Vol 1 and 2 (2007) 1114–1120.

[59] F. Qayum, R. Heckel, Local search-based refactoring as graph transformation, in: Search Based Software Engineering, 2009 1st International Symposium on, IEEE, 2009, pp. 43–46.

[60] C. L. Simons, J. Smith, P. White, Interactive ant colony optimization (iaco) for early lifecycle software design, Swarm Intelligence 8 (2) (2014) 139–157.

[61] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, M. Ó Cinnéide, Recommendation system for software refactoring using innovization and interactive dynamic optimization, in: Proceedings of the 29^{th} ACM/IEEE Int'l Conf. on Automated software engineering, ACM, 2014, pp. 331–336.

[62] T. J. Dea, Improving the performance of many-objective software refactoring technique using dimensionality reduction, in: International Symposium on Search Based Software Engineering, Springer, 2016, pp. 298–303.

[63] M. W. Mkaouer, M. Kessentini, M. Ó. Cinnéide, S. Hayashi, K. Deb, A robust multi-objective approach to balance severity and importance of refactoring opportunities, Empirical Software Engineering (2016) 1–34.