Multi-View Model Refactoring using a Multi-Objective Evolutionary Algorithm

Usman Mansoor¹, Marouane Kessentini¹, Philip Langer², Tanja Mayerhofer², Manuel Wimmer², Kalyanmoy Deb³

¹ University of Michigan, MI, USA
² Vienna University of Technology, Vienna, Austria
³ Michigan State University, MI, USA

firstname@umich.edu
lastname@big.tuwien.ac.at
kdeb@egr.msu.edu

COIN Report Number 2015017
Department of Electrical and Computer Engineering, Michigan State University, East Lansing, USA http://ww.egr.msu.edu/~kdeb/reports.shtml

Abstract. To improve the quality of software systems, one of the widely used techniques is refactoring defined as the process of improving the design of existing system by changing its internal structure without altering the external behavior. The majority of existing refactoring works focus mainly on the source code level. The suggestion of refactorings at the model level is more challenging due to the difficulty to evaluate: a) the impact of the suggested refactorings applied to a diagram on other related diagrams to improve the overall system quality, b) their feasibility, and c) inter-diagram consistency. We propose, in this paper, a novel framework that enables software designers to apply refactoring at the model level. To this end, we used a multi-objective evolutionary algorithm to find a trade-off between improving the quality of different diagrams at the same time such as class diagrams and activity diagrams. The proposed multi-objective approach provides a multi-view for software designers to evaluate the impact of suggested refactorings applied to class diagrams on related activity diagrams in order to evaluate the overall quality, and check their feasibility and behavior preservation. The statistical evaluation performed on models extracted from four open source systems confirms the efficiency of our approach.

1 Introduction

Model-Driven Engineering (MDE) considers models as first-class artifacts during the software lifecycle [19]. Available techniques, approaches, and tools for MDE are growing and they support a huge variety of activities, such as model creation, model transformation, and code generation. In the context of code generation, the quality of the generated source code from models depends on the quality of the source models. In addition, the maintainability and quality assurance goals are defined, in general, by software managers and team leads and they prefer to evaluate the quality of the software systems at the model level because it represents a better representation than
the source code to identify, suggest and evaluate different strategies to reach some maintainability objectives.

A widely used technique to improve the overall quality of systems is refactoring which improves design structure while preserving the overall functionalities and behavior [2]. A variety of refactoring work has been proposed in the literature [2][44] and the majority of them focus only on the source code level. Despite its importance, model refactoring is still in its teenage years [27][29][35]. In fact, model refactoring is more difficult and challenging than code refactoring for several reasons. First, the evaluation of the impact of refactorings in the model level is difficult to estimate. In the source code level, traditional code quality metrics are used to evaluate the quality of a system after applying a sequence of refactorings. However, applying refactoring on a specific model such as class diagrams have an impact on related other diagrams such as activity diagrams, sequence diagrams, etc. Sometimes, an improvement of class diagram quality metrics may decrease the quality of an activity diagram. Thus, it is important to evaluate the impact of suggested refactorings not only on one diagram but also other related diagrams to estimate the overall quality. Second, some refactorings suggested at the model level cannot be applied to the source code level. For example, a move method between two classes can be applicable at the class diagram level but cannot be applied in the source code one. Such situations can be detected using an activity diagram that can evaluate the feasibility of some refactorings. Third, it is difficult to check if a refactoring applied to a class diagram preserves the behavior or not without the use of some related behavioral diagrams such as an activity diagram.

To address these issues, we propose in this paper a model refactoring approach based on a multi-objective evolutionary algorithm, namely NSGA-II proposed by Deb [45]. Our multi-objective approach aims to find the best sequence of refactorings that provides a good trade-off between maximizing the quality of class diagrams and activity diagrams while preserving some behavioral constraints defined on activity diagrams. Our NSGA-II algorithm starts by generating a sequence of refactorings applied to a class diagram then we automatically generate the equivalent activity diagram using some co-evolution rules. The evaluation of the proposed refactorings is based on three objectives: a) maximizing a set of class diagram quality metrics; b) maximizing a set of activity diagram metrics and c) minimizing the number of violated behavioral preservation constraints. The paper reports on the results of an empirical study of our multi-objective proposal as applied to a set of models extracted from four open source systems. We compared our approach to a mono-objective genetic algorithm [46] and an existing technique, DesignImpl, not based on heuristic search [47].

The remainder of this paper is structured as follows. Section 2 provides the background of model refactoring and demonstrates the challenges addressed in this paper based on a motivating example. In Section 3, we give an overview of our proposal and explain how we adapted the NSGA-II algorithm to find optimal model refactoring sequences. Section 4 discusses the design and results of the empirical evaluation of our approach. After surveying related work in Section 5, we conclude with some pointers to future work in Section 6.
2 Model Refactoring Challenges

Finding an optimal sequence of refactorings on class diagrams and corresponding co-refactorings on activity diagrams in order to accomplish a high quality of both views on a software system is a challenging task, because the effects of refactorings may improve the quality of one view, while they decrease the quality of the other. In this section, we introduce some well-known quality metrics that we use to evaluate the overall quality of the design and discuss the refactorings of class diagrams and corresponding co-refactorings of activity diagrams that can be applied to improve the quality of both views. Based on these quality metrics and refactorings, we showcase the challenge of finding an optimal sequence based on a small example. However, at the same time we like to stress that our approach is not limited to these specific multi-view refactoring problem but it is usable as a general approach to tackle also other multi-view refactoring scenarios.

2.1 Quality Metrics

Several metrics have been proposed to evaluate the structural quality of software artifacts (e.g., [1]). Many of those metrics have also been successfully adopted for evaluating the structural design quality of UML (meta-)models, e.g., by Ma et al. [3]. Based on those works, we selected several metrics for class diagrams and activity diagrams (for activity diagrams we mostly based our metrics on existing work in the field of business processes, e.g., [5]) covering their design size and complexity (e.g., number of attributes and methods per class, number of parameters of methods, etc.), their coupling and encapsulation (e.g., number of associations, number data accesses over associations), as well as their abstraction (e.g., inheritance depth, number of polymorphic methods). A complete list of the used metrics and their definition is explained in Section 3.

2.2 Refactorings

The refactoring of object-oriented programs is a well-researched domain [2] and many of the identified refactorings for object-oriented programs have been adopted for the refactoring of design models [4]. In this paper, we consider those refactorings that are applicable on class diagrams and identified the necessary co-refactorings for activity diagrams. The co-refactoring of activities is necessary after applying a refactoring to the class diagram in order to maintain the validity of consistency rules among classes and activities. A complete list of the considered class diagram refactorings and the corresponding co-refactorings of activities is available in [48].

Consistency rules When using class diagrams and activity diagrams to represent the structure and behavior of programs, the consistency rules between class diagrams and activity diagrams largely correspond to the static semantics rules between classes and statements of an object-oriented, statically-typed programming language. To avoid ambiguities regarding the semantics of classes, activities, and actions, we adopt the
semantics of the Foundational Subset For Executable UML (fUML) to define consistency rules and to derive necessary co-refactorings. As an example for such consistency rules, we may consider a ReadStructuralFeatureValueAction in an activity, which obtains the value of a specific feature from an object. The consistency rule of this action with respect to the class diagram is that the feature to be read must be a direct or inherited feature of the object’s class. Moreover, the feature must be visible in the current context.

**Refactorings and Co-refactorings** We consider 15 well-known refactorings of class diagrams ranging from moving features, such as properties and operations, through extracting classes or superclasses from other classes, as well as pushing down and pulling up features along inheritance relationships, through to replacing inheritance with delegation and vice versa. For each of those refactorings of class diagrams, we identified the necessary co-refactorings for activity diagrams to maintain the validity of consistency rules between classes and activities. For instance, if a new class is extracted from one class and, thereby, a new association is added from the original class to the extracted class and one or more features (properties and operations) of the original class are moved to the new extracted class, all StructuralFeatureValueActions that access the moved features have to be prepended with a ReadStructuralFeatureValueAction that first reads the introduced association to navigate from the original class to the extracted class; otherwise, moved features would not be accessible in the object that is of the type of the original class. Note that in certain scenarios, it might not be possible to re-establish the validity of all conformance relationships with a co-refactoring of activities. For instance, when a private property of a class is pulled-up to its superclass and there are activities in the subclass reading this private property, we would have to pull-up this activity and the corresponding operation too. However, if this activity also reads other private properties that were not pulled up into the superclass, we cannot pull-up the activity and the operation; thus, it is not possible to establish valid conformance rules.

**2.3 Multi-View Refactoring Challenges and Motivating Example**

To showcase the challenges of finding an optimal sequence of refactorings to improve the quality of the class diagram and at the same time the quality of the activity diagram, consider the example illustrated in Figure 1. In this example, we have one class Circle containing three properties and two operations. In Figure 1, we also depict the activity diagram representing the behavior of the first operation distance(int, int). Besides, we show the measures of some of the quality metrics for this example, such as the number of properties per class (PPC), the number of edges and nodes in the activity, as well as its control-flow complexity (CFC), and its locality (LOC) – see legend of Figure 1 for their formulas.

As in our example, the class Circle contains two properties x and y, which specify the coordinates of its center point, we may apply the refactoring “Extract Class” to encapsulate these two parameters. Of course, we also have to co-refactor the activity diagram accordingly. The class and activity diagram after the refactoring and the co-refactoring is depicted in Figure 2. Thus, a new class Point has been introduced, which now contains the two properties x and y. Besides, a new association is created
to link the point from the class Circle. Alongside the class diagram, we had to apply co-refactorings in the operation distance(int, int). In particular, a new ReadStructuralFeatureValueAction ("point") has been added to obtain the values x and y. We may observe that, although the quality of the class diagram might have been improved (e.g., there is a better distribution of properties per class), the length, the number of edges, the control-flow complexity (CFC), as well as the locality (LOC) indicate a worse design with respect to the activity diagram. The reason for this is that the activity specifying the behavior of the operation distance(int, int) contains an additional read-action to obtain values from a referenced object.

**Figure 1 Motivating Example - Initial Version**

**Figure 2 Motivating Example – After Extract Class Point**

To improve this situation, we have to apply another refactoring, namely “Move Operation”, in order to move the operations distance(int, int) into the newly created class Point. Then, however, we break the conformance rules of class diagram and the
activity diagram, because in \texttt{distance(int, int)}, the operation \texttt{distance(int, int, int, int)} is called, which is not possible in the scope of \textit{Point}, since point has no access to the instance of \textit{Circle}. Nevertheless, when we accept the temporary inconsistency and also move the operation \texttt{distance(int, int, int, int)} into the class \textit{Point}, we obtain a new result, depicted in Figure 3, which not only validates all conformance rules, but also improves the metrics of the activity diagram significantly; the number of edges has been reduced and the control-flow complexity, as well as the locality, has been improved.

In conclusion, applying refactorings on the class diagram may have a strong impact on the quality of the activity diagrams that specify the behavior of the classes’ operations. Even worse, in several scenarios, the class refactorings will break their consistency. Finding a good sequence of refactorings to obtain a consistent and improved class and activity diagram is a major challenge. First, we have to deal with a multi-dimensional optimization problem, and second, we may have to accept temporarily inconsistencies to ultimately reach even better solutions.

![Activity Diagram](image)

**Figure 3 Motivating Example – After Move Methods distance**

### 3 Multi-View Model Refactoring

We describe, in this section, our multi-view approach for model refactoring. We start by giving an overview of our proposal and provide subsequently a more detailed description on how we adapted and used the NSGA-II algorithm for the problem of model refactoring.
3.1 Approach Overview

The goal of our approach is to generate the best refactoring sequence that improve the quality of different diagrams at the same time and also preserve some behavior preservation constraints. Therefore, we use a multi-objective optimization algorithm to compute an optimal sequence of refactorings in terms of finding trade-offs between maximizing the quality of class diagrams and activity diagrams, and minimizing the number of violated behavioral preservation constraints. In fact, the evaluation of refactorings applied on a class diagram depends on their impact on the related diagrams such as activity diagrams. In addition, activity diagrams should be used to verify if the behavior is changed after applying the refactorings on the class diagram.

The general structure of our approach is sketched in Figure 4. The search-based process takes as inputs the list of 15 possible types of refactoring that can be applied to a class diagram, the list behavioral preservation constraints, the co-evolution rules to generate the equivalent activity diagram from a refactored class diagram, a list of metrics to evaluate the quality of class diagrams and activity diagrams, and the system design to refactor. The process of generating a solution can be viewed as the mechanism that finds the best refactorings sequence among all possible solutions that minimizes the number of violated behavioral constraints, maximizes the quality of the class diagram and maximizes also the quality of the related activity diagram. The size of the search space is determined not only by the number of refactorings but also by the order in which they are applied. Due to the large number of possible refactoring combinations and the three objectives to optimize, we considered model refactoring as a multi-objective optimization problem. In the next subsection, we describe the adaptation of NSGA-II proposed by Deb et al. [45] to our problem domain.

![Figure 4. Multi-objective model refactoring: overview](image)

3.2 Multi-Objective Formulation

3.2.1 NSGA-II

An optimization problem consists in searching for an optimal or near-optimal solution within a predefined search space where the goal is to maximize or minimize a quality function called objective function. Differently to single-objective optimization problems where we are looking for a single optimal solution, the resolution of a multi-objective problem (MOP) yields a set of compromise solutions, called non-dominated solutions, and their image in the objective space is called the Pareto front. In what follows, we give some background definitions related to this topic:
Definition 1 (MOP). A MOP consists in minimizing or maximizing a set of objective functions under some constraints [45]. An MOP could be expressed as:

\[
\begin{aligned}
\min f(x) &= \{f_1(x), f_2(x), \ldots, f_M(x)\}^T \\
g_j(x) &\geq 0 \quad j = 1, \ldots, P; \\
h_k(x) &= 0 \quad k = 1, \ldots, Q; \\
x_i^L &\leq x_i \leq x_i^U \quad i = 1, \ldots, n.
\end{aligned}
\]

where \(M\) is the number of objective functions, \(P\) is the number of inequality constraints, \(Q\) is the number of equality constraints, \(x_i^L\) and \(x_i^U\) correspond to the lower and upper bounds of the variable \(x_i\). A solution \(x\) satisfying the \((P+Q)\) constraints is said feasible and the set of all feasible solutions defines the feasible search space denoted by \(\Omega\). In this formulation, we consider a minimization MOP since maximization can be easily turned to minimization based on the duality principle by multiplying each objective function by \(-1\). The resolution of a MOP consists in approximating the whole Pareto front.

Definition 2 (Pareto optimality). A solution \(x^* \in \Omega\) is Pareto optimal if there does not exist any solution \(x\) such that \(f_m(x) < f_m(x^*)\) for all \(m\).

The definition of Pareto optimality states that \(x^*\) is Pareto optimal if no feasible vector \(x\) exists which would improve some objective without causing a simultaneous worsening in at least another one. Other important definitions associated with Pareto optimality are essentially the following:

Definition 3 (Pareto dominance). A solution \(u = (u_1, u_2, \ldots, u_n)\) is said to dominate another solution \(v = (v_1, v_2, \ldots, v_n)\) (denoted by \(f(u) \preceq f(v)\)) if and only if \(f(u)\) is partially less than \(f(v)\). In other words, \(\forall m \in \{1, \ldots, M\}\) we have \(f_m(u) \leq f_m(v)\) and \(\exists m \in \{1, \ldots, M\}\) where \(f_m(u) < f_m(v)\).

Definition 4 (Pareto optimal set). For a MOP \(f(x)\), the Pareto optimal set is \(P^* = \{x \in \Omega : \exists x' \in \Omega, f(x') \preceq f(x)\}\).

Definition 5 (Pareto optimal front). For a given MOP \(f(x)\) and its Pareto optimal set \(P^*\), the Pareto front is \(PF^* = \{f(x), x \in P^*\}\).

Several methods were proposed in the literature to solve MOPs [3]. Due to their population-based nature, Evolutionary Algorithms (EAs) have shown their effectiveness and efficiency in providing a well-converged and well-diversified approximation of the Pareto front [4] independently of its geometrical nature which is not the case for classical mathematical methods. Among the most used Multi-Objective EAs (MOEAs), we cite NSGA-II, SPEA2, IBEA and MOEA/D. Since the most used MOEA within the SBSE community is NSGA-II [45], we choose to use it in this study.

NSGA-II is one of the most used and effective MOEAs. It begins by generating an offspring population from a parent one by means of variation operators (crossover and mutation) such that both populations have the same size. After that, it ranks the
merged population (parents and children) into several non-dominated layers, called fronts, as depicted by Figure 5. Non-dominated solutions are assigned a rank of 1 and constitute the first layer. Non-dominated solutions according to the population truncated of the layer 1 are assigned a rank of 2 and constitute the layer 2. This process is continued until the ranking of all parent and children individuals. After that, each solution is assigned a diversity score, called crowding distance, frontwise. This distance corresponds to the half of the perimeter of the cuboid having the two closest neighboring solutions to the considered individual as vertices. It is important to note that extreme solutions are assigned an infinite crowding score since they are of great importance for diversity. The fitness in NSGA-II is not a scalar value. In fact, it is a couple \((\text{rank}, \text{crowding distance})\). Solutions having better ranks are emphasized. Among solutions having the same rank (belonging to the same layer), solutions having larger crowding distances are emphasized since they are less crowded than the others. Once all individuals of the merged population are assigned a rank and a diversity score, we perform the environmental selection to form the parent population for the next generation. Indeed, solutions belonging to the best layers are selected. Figure 1 illustrates this process where the last selected layer is the 4\(^{th}\) one. Usually, the cardinality of the last layer (layer 4 in figure 1) is greater than the number of available slots in the parent population of the next generation. As denoted by figure 1, solutions of the 4\(^{th}\) layer are selected based on their crowding distance values. In this way, most crowded solutions are discouraged to remain in the race; thereby emphasizing population diversification. To sum up, the Pareto ranking encourages convergence and the crowding factor procedure emphasizes diversity, therefore NSGA-II is an elitist multi-objective EA which is today the most used metaheuristic in multi-objective applied optimization.

![Figure 5](image)

**Figure 5** NSGA-II replacement scheme for a bi-objective maximization case.

### 3.2.2 NSGA-II Adaptation for Model Refactoring

This section describes how NSGA-II [45] can be used to find design refactoring solutions with multiple conflicting objectives. To apply NSGA-II to a specific problem, the following elements have to be defined:

- Representation of the individuals;
- Evaluation of individuals using a fitness function for each objective to optimize to determine a quantitative measure of their ability to solve the problem under consideration;
- Selection of the individuals to transmit from one generation to another;
• Creation of new individuals using genetic operators (crossover and mutation) to explore the search space.

Next, we describe the adaptation of the design of these elements for the generation of model refactoring solutions using NSGA-II.

a) Solution representation

To represent a candidate solution (individual), we used a vector representation. Each vector’s dimension represents a class diagram refactoring operation. Thus, a solution is defined as a long sequence of refactorings applied to different parts of the design. When created, the order of applying these refactorings corresponds to their positions in the vector. In addition, for each refactoring, a set of controlling parameters (stored in the vector), e.g., actors and roles are randomly picked from the class diagram to be refactored and stored in the same vector. An example of a solution is given in Figure 6 on the class diagram of the motivating example of Figure 2.

![Figure 6](image-url)

Figure 6. Representation of an NSGA-II individual

After the generation of the refactoring for the class diagram, we automatically generate the equivalent activity diagram refactorings using the co-evolution rules described in Section 2. These activity diagram refactorings are also represented in a vector similar to those applied to the class diagram. Moreover, when creating a sequence of refactorings (individuals), it is important to guarantee that they are feasible and that they can be legally applied. Some of these behavior preservation constraints are defined in both class diagrams and activity diagrams as described in Section 2. For example, to apply the refactoring operation move method(Circle, Point, distance(int, int)), a number of necessary preconditions should be satisfied, e.g., Circle and Point should exist and should be classes; distance() should exist and should be a method; the classes Circle and Point should not be in the same inheritance hierarchy; the method distance() should be implemented in Circle; the method signature of distance() should not be present in class Point. As postconditions, Circle, Point, and distance() should exists; distance() declaration should be in the class Point; and distance() declaration should not exists in the class Circle. Other constraints checked by the activity diagram are discussed in Section 2.

b) Fitness functions

After creating a solution, it should be evaluated using fitness functions. Since we have three objectives to optimize, we are using three different fitness functions to include in our NSGA-II adaptation. We used the following fitness functions:

1. Quality of the class diagram fitness function is calculated using a set of 11 quality metrics used by the QMOOD model [49] described in Table 1. All the 11 metrics are aggregated in one fitness function with equal importance and normalized between 0 and 1.

![Table 1](image-url)

Table 1 QMOOD metrics for design properties [49]
<table>
<thead>
<tr>
<th>Complexity</th>
<th>NOM</th>
<th>Number of methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coupling</td>
<td>DCC</td>
<td>Direct class coupling</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>NOP</td>
<td>Number of polymorphic methods</td>
</tr>
<tr>
<td>Hierarchies</td>
<td>NOH</td>
<td>Number of hierarchies</td>
</tr>
<tr>
<td>Cohesion</td>
<td>CAM</td>
<td>Cohesion among methods in class</td>
</tr>
<tr>
<td>Abstraction</td>
<td>ANA</td>
<td>Average number of ancestors</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>DAM</td>
<td>Data access metric</td>
</tr>
<tr>
<td>Composition</td>
<td>MOA</td>
<td>Measure of aggregation</td>
</tr>
<tr>
<td>Inheritance</td>
<td>MFA</td>
<td>Measure of functional abstraction</td>
</tr>
<tr>
<td>Messaging</td>
<td>CIS</td>
<td>Class interface size</td>
</tr>
</tbody>
</table>

2. *Quality of the activity diagram fitness function* represents an aggregation (sum) of 12 metrics: Number of parameters, Number of nodes, Number of edges, Number of actions, Control-Flow Complexity (CFC), Interface Complexity, Halstead-based Activity Complexity, Coefficient of Network Complexity, Fan-in/Fan-out Metrics for Activities, Tree depth metric, Tree width metric and Locality. All these metrics are normalized between 0 and 1.

3. *Number of violated behavioral constraints fitness function* checks how many behavioral constraints are violated by the generated refactoring solutions when applied to an activity diagram. These constraints are described in Section 2.

c) **Selection**

To guide the selection process, NSGA-II uses a binary tournament selection based on dominance and crowding distance [45]. NSGA-II sorts the population using the dominance principle which classifies individual solutions into different dominance levels. Then, to construct a new offspring population $Q_{t+1}$, NSGA-II uses a comparison operator based on a calculation of the crowding distance to select potential individuals having the same dominance level.

d) **Genetic operators**

To better explore the search space, the crossover and mutation operators are defined. For crossover, we use a single, random, cut-point crossover. It starts by selecting and splitting at random two parent solutions. Then crossover creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and, for the second child, the first part of the second parent with the second part of the first parent. This operator must ensure that the length limits are respected by eliminating randomly some refactoring operations. As illustrated in Figure 7, each child combines some of the refactoring operations of the first parent with some ones of the second parent. In any given generation, each solution will be the parent in at most one crossover operation.

The mutation operator picks randomly one or more operations from a sequence and replaces them by other ones from the initial list of possible refactorings. An example is shown in Figure 8. After applying genetic operators (mutation and crossover), we verify the feasibility of the generated sequence of refactoring by checking the pre and post conditions. Each refactoring operation that is not feasible due to unsatisfied preconditions will be removed from the generated refactoring sequence. The new sequence is considered valid in our NSGA-II adaptation if the number of rejected refactorings is less than 5% of the total sequence size.
Overall, the adaptation of NSGA-II to our model refactoring problem is generic thus it can be easily extended to include other modelling languages by adding a new fitness function (to evaluate the quality of the new type of models). The solution representation and change operators will remain the same. Of course, the input should be also extended to integrate new quality metrics related to the new considered modelling language that will be used by the new fitness function as a new objective to optimize.

4 Validation

In order to evaluate the feasibility and the efficiency of our approach for generating good design refactoring suggestions, we conducted an experiment based on different versions of open-source systems. We start by presenting our research questions. Then, we describe and discuss the obtained results.

4.1 Research Questions

In our study, we assess the performance of our model refactoring approach by finding out whether it could generate meaningful sequences of refactorings that improve the structure of class diagrams and activity diagrams while preserving the behavior. Our study aims at addressing the following research questions outlined below. We also explain how our experiments are designed to address these questions. The main question to answer is to what extent can the proposed approach proposes meaningful model refactoring solutions. To find an answer, we defined the following research questions:

RQ1: To what extent can the proposed approach improves the quality of class diagrams and activity diagrams?

RQ2: To what extent the proposed approach preserves the behavior while improving the quality?

RQ3: How does the proposed multi-objective approach based on NSGA-II perform compared to a mono-objective approach where only one objective is considered to improve the quality of class diagrams?

RQ4: How does the proposed multi-objective design refactoring approach perform compared to existing model refactoring approach [47] not based on heuristic search?
RQ5: Insight. How our multi-objective model refactoring approach can be useful for software engineers in real-world setting?

To answer RQ1, we validate the proposed design refactoring solutions to improve the quality of the system by evaluating their ability to fix some design defects that can be detected on class diagrams extracted from four open-source systems. We adapted our previous work [50] based on quality metrics rules to detect three types of design defects: Blob (it is found in designs where one large class tends to centralize the functionalities of a system, and the other related classes primarily exposing data.), Long Parameter List (methods with numerous parameters are a challenge to maintain, especially if most of them share the same data-type) and Data Clumps (interrelated data items which often occur as clump in the model. The same data items are often together in different places such as attributes in classes or parameters in method signatures). We defined a measure NFD that corresponds to the ratio of the number of corrected design defects over the initial number of detected defects on a class diagram before applying the suggested refactoring solution.

\[
NFD = \frac{\text{#fixed design defects on a class diagram}}{\text{#defects before applying refactorings}}
\]

It is also important to assess the refactoring impact on the design quality and not only on a class diagram. The expected benefit from refactoring is to enhance the overall software design quality as well as fixing design defects. The quality metrics considered by our approach can improve different aspects of the design quality related to reusability, flexibility, understandability, functionality, extendibility, and effectiveness. The improvement in quality can be assessed by comparing the quality before and after refactoring independently to the number of fixed design defects. Hence, the total gain in quality \( G \) before and after refactoring can be easily estimated as:

\[
G = \frac{1}{23} \sum_{i=1}^{23} [q'_i - q_i],
\]

where \( q'_i \) and \( q_i \) represents the value of the quality attribute \( i \) respectively after and before refactoring. As described in the previous section, we considered a total of 23 metrics related to both class diagrams and activity diagrams.

To answer RQ2, we asked groups of potential users of our refactoring tool to evaluate, manually, whether the suggested refactorings are feasible and preserve the behavior or not. We define the metric “refactoring precision” (RP) which corresponds to the number of meaningful refactorings over the total number of suggested refactoring operations:

\[
RP = \frac{\text{#feasible refactorings}}{\text{#proposed refactorings}}
\]

To answer RQ3, we compare our approach to a mono-objective formulation using a genetic algorithm (GA) that considers the refactoring suggestion task only from the class diagram quality improvement perspective (single objective).

To answer RQ4, we compared our design refactoring results with a recent tool, called Design-Impl proposed recently by Iman and Mel [47] that do not use heuristic search techniques. The current version of Design-Impl is implemented as an Eclipse plug-in that proposes a list of class diagram and code refactorings based on an
interaction with the designer who specify the desired design based on an evaluation of the class diagram.

To answer RQ5, we asked a group of eight software engineers to refactor manually some of the detected design defects on the class diagrams, and then compare the results with those proposed by our tool. To this end we define the following precision metric $MP$ (Manual precision): $MP = \frac{|R \cap R_m|}{R}$, where $R$ is the set of refactorings suggested by our tool, and $R_m$ is the set of refactorings suggested manually by software engineers.

### 4.2 Experimental Settings

Our study considers model fragments extracted from four open source projects: GanttProject (Gantt for short), Rhino, JFreeChart and Xerces-J. Xerces-J is a family of software packages for parsing XML. GanttProject is a cross-platform tool for project scheduling. JFreeChart is a powerful and flexible Java library for generating charts. Finally, Rhino is a JavaScript interpreter and compiler written in Java and developed for the Mozilla/Firefox browser. Table 1 summarizes for each model the number of detected design defects using our previous work [50], as well as the number of model elements. We selected these systems for our validation because they range from medium to large-sized open source projects that have been actively developed over the past 10 years, and include a large number of design defects. Our study involved 6 subjects from the University of Michigan and some of them are working in automotive industry companies. Subjects include 4 master students in Software Engineering and 2 PhD students in Software Engineering. 4 of them are working in industry as senior software engineers. All the subjects are volunteers and familiar with Java development. The experience of these subjects on Java programming ranged from 6 to 16 years.

Parameter setting has a significant influence on the performance of a search algorithm on a particular problem instance. For this reason, for each algorithm and for each system, we perform a set of experiments using several population sizes: 50, 100, 200, 300 and 500. The stopping criterion was set to 100000 evaluations for all algorithms in order to ensure fairness of comparison. The other parameters’ values were fixed by trial and error and are as follows: (1) crossover probability = 0.8; mutation probability = 0.5 where the probability of gene modification is 0.3; stopping criterion = 100000 evaluations.

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is performed based on 51 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 99% confidence level ($\alpha = 1\%$). The latter verifies the null hypothesis $H_0$ that the obtained results of two algorithms are samples from continuous distributions with equal medians, as against the alternative that they are not, $H_1$. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis $H_0$ while it is true (type I error). A p-value that is less than or
equal to $\alpha (\leq 0.05)$ means that we accept $H_1$ and we reject $H_0$. However, a p-value that is strictly greater than $\alpha (> 0.05)$ means the opposite. In fact, for each problem instance, we compute the p-value of mono-objective search results with NSGA-II ones. In this way, we could decide whether the outperformance of NSGA-II over one of each of the others (or the opposite) is statistically significant or just a random result.

We note that the mono-objective algorithm provides only one refactoring solution, while NSGA-II generates a set of non-dominated solutions. In order to make meaningful comparisons, we select the best solution for NSGA-II using a knee point strategy [51]. The knee point corresponds to the solution with the maximal trade-off between the three objectives. Hence moving from the knee point in either direction is usually not interesting for the user. We use the trade-off “worth” metric proposed by Rachmawati and Srinivasan [51] to find the knee point. This metric estimates the worthiness of each non-dominated merging solution in terms of trade-off between our three conflicting objectives. After that, the knee point corresponds to the solution having the maximal trade-off “worthiness” value.

### 4.3 Results

**Results for RQ1.** As described in Figure 8, after applying the proposed refactoring operations by our approach (NSGA-II), we found that, on average, more than 85% of the detected design defects (model smells) were fixed ($NFD$) for all the class diagrams extracted from the four studied systems. This high score is considered significant to improve the quality of the refactored diagrams by fixing the majority of defects that were from different types. We found that the majority of non-fixed defects are related to the blob type. This type of defect usually requires a large number of refactoring operations and is known to be very difficult to fix.

<table>
<thead>
<tr>
<th>Systems</th>
<th>Release</th>
<th>#Elements</th>
<th>#Smells</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFreeChart</td>
<td>v1.0.9</td>
<td>81</td>
<td>22</td>
</tr>
<tr>
<td>GanttProject</td>
<td>v1.10.2</td>
<td>114</td>
<td>28</td>
</tr>
<tr>
<td>Xerces-J</td>
<td>V2.7.0</td>
<td>96</td>
<td>31</td>
</tr>
<tr>
<td>Rhino</td>
<td>v1.7R1</td>
<td>88</td>
<td>26</td>
</tr>
</tbody>
</table>

**Table 2.** Systems studied

![Figure 8. NFD median values of NSGA-II, GA and DesignImpl over 51 independent systems.](image-url)
simulation runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).

In Figure 9, we show the obtained gain values (in terms of absolute value) that we calculated for all the metrics considered for both class diagrams and activity diagrams before and after refactoring for each studied system. We found that the diagrams quality increases across all the quality factors. Furthermore, we noticed that the quality of both class diagrams and activity diagrams is improved. The highest quality improvement scores on all systems are mainly observed on class diagrams. To sum up, we can conclude that our approach succeeded in improving the design quality not only by fixing the majority of detected model smells but also by improving the user understandability, the reusability, the flexibility, as well as the effectiveness of the refactored design.

Results for RQ2. Regarding the behavior preservation and the feasibility of suggested refactorings, for all of the studied diagrams, an average of more than 80% of proposed refactoring operations are considered as feasible and do not generate behavior incoherence as described in Figure 10. A slight loss in the NFD and $G$ is largely compensated by the significant improvement in terms of refactorings feasibility and behavior preservation.

Figure 9. Design quality improvements ($G$) median values of NSGA-II, GA and DesignImpl over 51 independent simulation runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).

Figure 10. The refactoring precision (RP) median values of NSGA-II, GA and DesignImpl over 51 independent simulation runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).
Results for RQ3. As described in Figures 8, 9 and 10, it is clear that our proposal outperforms both the mono-objective GA and DesignImpl. Figures 8 and 9 show that our approach improves the quality of the design with a comparable values to both GA and DesignImpl. However, in terms of behavior preservation it is clear that our approach provides much more feasible refactorings than GA and designImpl for all the systems considered in our experiments. This can be explained by the fact that our proposal checks the behavior preservation using the activity diagrams however existing approaches did not consider it.

Results for RQ4. To evaluate the relevance of our suggested design refactorings for real software engineers, we compared the refactoring strategies proposed by our technique and those proposed manually by the subjects (software engineers) to fix several model smells on the diagrams considered in our experiments. Figure 11 shows that most of the suggested refactorings by NSGA-II are similar to those applied by developers with an average of more than 70%. Some defects can be fixed by different refactoring strategies and also the same solution can be expressed in different ways. Thus, we consider that the average of precision of more than 70% confirm the efficiency of our tool for real developers to automate the refactoring process. It is clear that our proposal outperforms GA and DesignImpl on all the diagrams, this can be explained by the fact that both of these techniques do not consider the behavioral constraints defined on the activity diagrams.

Figure 11. The MP median values of NSGA-II, GA and DesignImpl over 51 independent simulation runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).

Another advantage related to the use of our multi-objective approach is the diversity of non-dominated solutions as described in Figure 12. Figure 12 depicts the Pareto front obtained on Xerces using NSGA-II to optimize the three considered objectives. Similar fronts were obtained on the remaining systems. The 2-D projection of the Pareto front helps software engineers to select the best trade-off solution between the three objectives based on their own preferences. Based on the plots of Figure 12, the engineer could degrade quality in favor of behavior preservation. In this way, the user can select the preferred refactoring solution to realize. This is a very interesting feature, since recent studies [2] showed that software developers still select refactoring solutions that could change the behavior with a high quality improvement because they believe that it is easy to fix the behavior violation.
4.4 Threats to Validity

We explore, in this section, the factors that can bias our empirical study. These factors can be classified in three categories: construct, internal, and external validity. Construct validity concerns the relation between the theory and the observation. Internal validity concerns possible bias with the results obtained by our proposal. Finally, external validity is related to the generalization of observed results outside the sample instances used in the experiment.

In our experiments, construct validity threats are related to the absence of similar work that uses multi-objective techniques for automated multi-view model refactoring. For that reason, we compare our proposal with GA-based approach and an existing semi-automated design refactoring technique. Another threat to construct validity arises because, although we considered 3 types of model smells, we did not evaluate the performance of our proposal with other model smell types. In future work, we plan to use additional model smell types and evaluate the results.

We take into consideration the internal threats to validity in the use of stochastic algorithms since our experiment is performed based on 51 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 99% confidence level ($\alpha = 1\%$). However, the parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work by additional experiments.

External validity refers to the generalization of our findings. In this study, we performed our experiments on diagrams extracted from different widely-used open-source systems belonging to different domains and with different sizes. However, we cannot assert that our results can be generalized to other applications, and to other practitioners. Future replications of this study are necessary to confirm the general aspect of our findings.

5 Related Work

With respect to the contribution of this work, we organize related approaches using three categories of related work: (i) refactoring approaches working solely on the
model level, (ii) refactoring working on model and code level that may be also considered as a kind of multi-view refactoring, and (iii) widely related approaches working solely on the code level.

Model refactorings. Two surveys concerning model refactorings are available [34][39] that discuss different research trends and classifications for model refactoring. One of the first investigations in this area was done by Sunyé et al. [21] who define a set of UML refactorings on the conceptual level by expressing pre- and post-conditions in OCL. Boger et al. [7] present a refactoring browser for UML supporting the automatic execution of pre-defined UML refactorings. While these two approaches focus on pre-defined refactorings, approaches by Porres [19], Zhang et al. [22], and Kolovos et al. [16] discuss the introduction of user-defined refactorings by using dedicated textual languages for their implementation. A similar idea is followed in [6][18] where graph transformations are used to describe refactorings and graph transformation theory is applied for analyzing model refactorings. Pattern-based refactoring for UML models with model transformations is presented in [42].

The mentioned approaches cover mostly single-view refactorings and focus on the implementation of semi-automatically executable refactorings. Only some approaches for tackling consistency between different views in the context of refactorings have been presented. For instance, [17][33] proposed to refactor UML class diagrams, also adapting attached OCL constraints. Another approach that considers the effect on refactorings of UML class diagrams on operations implemented in OCL with respect to behavioral equivalence is presented in [41]. A constraint-based refactoring approach for UML is presented in [36] which considers well-formedness rules and translates refactorings to CSP to eventually compute the additional changes required for a semantic-preserving model refactoring.

Reuse of model refactorings for different languages is discussed in [31] by specifying generic role-based refactorings that can be bound to specific languages. Another approach aiming for generic model refactorings is present in [37] by using a combination of aspect weaving and model typing. Refactorings are developed on a generic metamodel and may be reused for specific metamodels which fulfill the model typing relationship to the generic metamodel.

In [32] tool support for defining model metrics, smells, and refactorings is presented. In particular, language specific and project specific metrics, smells, and refactorings for the design-level may be defined based on graph transformations. A refactoring approach considering performance optimization of models, i.e., runtime-level, is presented in [35]. In this context, refactorings are used to eliminate anti-patterns that may have a negative impact on performance aspects.

Related to multi-view refactoring is the field of multi-view consistency [12]. We have early works on multi-view consistency [16][18] using a generic representation of modifications and relying on users to write code to handle each type of modification in each type of view. This idea influenced later efforts on model synchronization frameworks in general [24][25] and in particular bi-directional model transformations [38][42]. Other approaches use so-called correspondence rules for synchronizing models in the contexts of RM-ODP and model-driven web engineering [8][15][37]. All these approaches have in common that they consider only atomic changes when reconciling models and not refactorings. In previous work [37], we presented coupled transformations to refactor different views altogether by automatically executed the coupled transformations when initial transformations are executed. Another work we are aware of allowing the propagation of more complex changes such as refactorings is [20] based on a kind of event/condition/action rules.

In previous work [40] we have proposed to use Interactive Genetic Algorithm (IGA) for model refactoring allowing the modelers to provide feedback during refactoring focusing on single-view improvements.
To sum up, all these mentioned model refactoring approaches are mostly considering a single view during refactoring. If multiple views are considered by approaches from multi-view synchronization, the only quality aspect that is taken care of is having consistency between the different viewpoints. To the best of our knowledge, our approach is the first one that considers multi-view model refactoring as an optimization problem for finding an optimal refactoring solution considering quality criteria for different views at once.

Model/Code Refactorings. The synchronization of models and code is of course also a challenging issue when it comes to refactoring. In [43], distributed graph transformations are used to specify coupled refactorings on UML models and Java code. Van Gorp et al. [25] have presented an extension of the UML metamodel which allows expressing the pre- and postconditions for refactorings as well as for representing method implementations in UML class diagrams based on the UML action semantics – a predecessor of fUML. Furthermore, they use OCL to detect code smells on the model level and propose to refactor designs independent of the underlying programming language on the model level by applying the following transformation chain: reverse engineering, model refactoring, and forward engineering. The approach for constraint-based model refactoring [36] discussed before, has been also extended for dealing model/code co-refactoring by so-called bridge constraints that capture the correspondences between model elements and the code elements [30].

To sum up, there are some approaches that consider refactorings on both model and code level, but we are not aware of any approach considering the models aligned with code as a multi-objective optimization problem.

Code Refactorings. Harman et al. [...] have proposed a search-based approach using Pareto optimality that combines two quality metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class), in two separate fitness functions. The authors start from the assumption that good design quality results from good distribution of features (methods) among classes. Their Pareto optimality-based algorithm succeeded in finding good sequence of move method refactorings that should provide the best compromise between CBO and SDMPC to improve code quality. However, one of the limitations of this approach is that it is limited to unique refactoring operation (move method) to improve software quality and only two metrics to evaluate the performed improvements. Recently, Ó Cinnéide et al. [...] have proposed a multi-objective search-based refactoring to conduct an empirical investigation to assess some structural metrics and to explore relationships between them. To this end, they have used a variety of search techniques (Pareto-optimal search, semi-random search) guided by a set of cohesion metrics. The main problem in all of these code-level approaches is that the consistency preservation are not considered to obtain correct and meaningful refactorings.

6 Conclusion

This paper presented a novel multi-view refactoring approach taking into consideration multiple criteria to suggest good and feasible design refactoring solutions to improve the design quality. The suggested refactorings preserve the behavior of the design to restructure and consider the impact of refactoring a class diagram on related diagrams such as activity diagrams. Our search-based approach succeeded to find the best trade-off between these multiple criteria. Thus, our proposal produces more meaningful refactorings in comparison to some of those
discussed in the literature [47]. Moreover, the proposed approach was empirically evaluated on several diagrams extracted from four open-source systems, and compared successfully to an existing approach not based on heuristic search [47].

In future work, we are planning to investigate an empirical study to consider additional views when suggesting refactorings such as sequence diagrams as well as object diagrams. We are also planning to consider a larger set of refactoring operations to fix additional types of model-smells.

References


[48] www.sbse.us/SQJ/

