Software Qual J (2017) 25:473–501
DOI 10.1007/s11219-015-9284-4

Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm

Usman Mansoor[1] · Marouane Kessentini[1] · Manuel Wimmer[2] · Kalyanmoy Deb[3]

Published online: 5 August 2015

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 an existing system by changing its internal structure without altering the external behavior. The majority of existing refactoring work focuses 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) interdiagram 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 class 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.

Keywords Search-based software engineering · Software maintenance · Multi-objective optimization

✉ Marouane Kessentini
marouane@umich.edu

Usman Mansoor
umansoor@umich.edu

Manuel Wimmer
wimmer@big.tuwien.ac.at

Kalyanmoy Deb
kdeb@egr.msu.edu

[1] University of Michigan, Dearborn, MI, USA

[2] Vienna University of Technology, Vienna, Austria

[3] Michigan State University, East Lansing, MI, USA

1 Introduction

Model-driven engineering (MDE) considers models as first-class artifacts during the software lifecycle (Porres 20052005). The 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 (Fowler et al. 1999). A variety of refactoring work has been proposed in the literature (Fowler et al. 1999; Mens and Tourwé 2004), and the majority of them focus only on the source code level. Despite its importance, model refactoring is still in its teenage years (Mens 2006; Mens et al. 2007b; Arcelli et al. 2012). 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 has an impact on related other diagrams such as activity diagrams and sequence diagrams. 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. Third, it is difficult to check whether 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 et al. (2002). Our multi-objective approach aims to find the best sequence of refactorings that provide a good trade-off between maximizing the quality of class diagrams and activity diagrams while preserving 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 (Goldberg 1989) and an existing technique, DesignImpl, not based on heuristic search (Moghadam and Cinneide 2012).

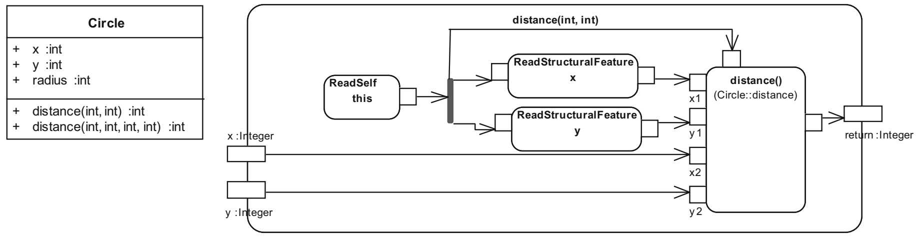
The remainder of this paper is structured as follows. Section 2 provides the background of model refactoring, and demonstrates the challenges addressed in this paper based on a motivating example. In Sect. 3, we give an overview of our proposal and explain how we adapted the NSGA-II algorithm to find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

2 Model refactoring challenges

Finding an optimal sequence of refactorings on class diagrams and the 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 the 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 maybe it can be used as a general approach to tackle also another multi-view refactoring scenarios.

2.1 Quality metrics

Several metrics have been proposed to evaluate the structural quality of software artifacts (e.g., Fenton and Pfleeger 1997). Many of those metrics have also been successfully adapted for evaluating the structural design quality of UML (meta-) models, e.g., by Ma et al. (2004). 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., (Cardoso 2006)] 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 Sect. 3.



Classes	
# classes	= 1
PPC	= 3
OPC	= 2

Activity	
# edges	= 9
# nodes	= 5
# actions	= 4
CFC	= 3
LOC	= 1

Legend
PPC … # properties per class
OPC … # operations per class,
CFC … control-flow complexity = #forks + # decisions
LOC … locality = # reads in referenced class / all reads

Fig. 1 Motivating example—initial version

Table 1 The list of refactorings and corresponding co-refactorings

Name of refactoring	Description	Co-evolution process
Rename class	Changes the name of a class with a new name, and updates its references	No co-refactoring needed, because in the abstract syntax the element references do not break with renames
Replace inheritance with delegation	Replaces a direct inheritance relationship with a delegation relationship	We assume that a delegation relationship is a 1..1 association For each operation of the original super class, introduce a new operation in the original subclass and activities that navigate through the delegation association (ReadStructuralFeatureValueAction) and call the respective operation (CallOperationAction) (Bock 2003) Existing CallOperationActions calling the original superclass's operation have to be changed in order to call the new introduced operation in original subclass instead ReadIsClassifiedObjectActions (instance of) have to be adapted, because the original subclass is not a subclass of the original superclass anymore
Replace delegation with inheritance	Replaces a delegation relationship with a direct inheritance relationship	Delete operations and activities from new subclass Existing CallOperationActions must be changed to call the operation of the new superclass instead
Extract subclass	Adds a new subclass to class C and moves the relevant features to it	The CreateObjectAction that created the object of class C must be changed to the create an object of the new subclass instead, if features that are pushed down in this refactoring are used on the created object Types of operation parameters have to be changed to the new subclass if the respective operation accesses features which are now only available in the subclass For methods being pushed down to the new subclass, see push down method/field
Extract superclass	Adds a new super class to class C and moves the relevant features to it	For the features being pulled up to the new superclass, see pull up method/field Co-evolution would be beneficial for code quality; the superclass should be used instead of the subclass wherever it is possible; that is, where no features are used that remain in the respective subclass
Collapse hierarchy	Removes a class from an inheritance hierarchy	Adaptation necessary; see extract superclass and extract subclass
Inline class	Moves all features of a class into another class and deletes it	We assume that a 1..1 association to the deleted class exists Access to attributes and call of operations have to be adapted (no navigation through association using a ReadStructuralFeatureValueAction is needed anymore) Adapt all references to deleted class and use class containing all its features instead in CreateObjectAction, ReadIsClassifiedObjectAction, parameter types, etc.

Table 1 continued

Name of refactoring	Description	Co-evolution process
Extract class	Creates a new class and moves the relevant features from the old class into the new one	We assume that a 1..1 association to the new class is introduced Delegating operations have to be added for operations moved to extracted class Read/Add/ClearStructuralFeatureValueActions and CallOperationValueActions have to be adapted. Before these can be executed, the object of the extracted class has to be obtained first through a ReadStructuralFeatureValueAction on the association pointing to the extracted class Usages of non-encapsulated and non-private attributes outside of the class from which the features were extracted having to be adapted (navigation to extract object has to be added) CreateObjectAction and respective linking for the extracted class has to be added wherever the existing class was instantiated (also DestroyObjectAction for new class has to be added)
Push down method	Moves a method from a class to those subclasses that require it	If the pushed-down operations were pushed down into multiple subclasses, these operations are moved *only from to one subclass* and *copied* from the other subclasses; thus, for the references to those operations must be adapted in all CallOperationActions depending on the type of the object on which the operation is called It must be ensured that the moved operations still have access to the used features (i.e., private attributes and operations in the superclass C must not be used in Read/Add/ClearStructuralFeatureValueActions, CallOperationAction, etc. in the moved operations) Pushed-down operations must be non-private, if they are calling somewhere in the superclass or on the level of the superclass type, because otherwise the pushed-down operations would not be accessible anymore If clients of the superclass call the operation, they must use the/a subclass instead (thus CreateObjectActions or parameter types must be adopted)
Pull up method	Moves a method of some class (es) to the immediate superclass	If the pulled-up operations were pulled up from multiple subclasses, these operations are moved *only from one class* and *removed* from the other subclasses; thus, for all CallOperationActions that point to the removed operation, the corresponding moved operation in the new superclass has to be used instead of the deleted ones Pulled-up operations must be non-private, if they are used somewhere in the subclass, because otherwise they would not be accessible anymore from the subclasses
Rename method	Changes the name of a method to a new one, and updates its references	No co-refactoring needed

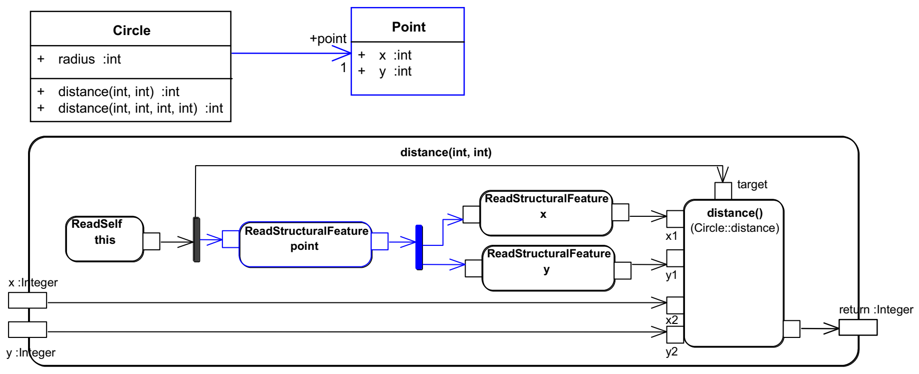
Table 1 continued

Name of refactoring	Description	Co-evolution process
Push down field	Moves a field from a class to those subclasses that require it	If the pushed-down fields were pushed down to multiple subclasses, these fields are moved *only from to one subclass* and *copied* from the other subclasses; thus, for the references to those fields must be adapted in all StructuralFeatureValueActions depending on the type of the object on which the field is accessed Pushed-down fields must be non-private, if they are accessed somewhere in the superclass or on the level of the superclass type, because otherwise the pushed-down fields would not be accessible anymore If clients of the superclass access the pushed-down fields, they must use the subclass instead (thus CreateObjectActions or parameter types must be adopted)
Pull up field	Moves a field from some class(es) to the immediate superclass	If the pulled-up fields were pulled up from multiple subclasses, these features are moved *only from one class* and *removed* from the other subclasses; thus, for all actions that access those removed features, the respective corresponding moved field in the new superclass has to be used instead of the deleted ones in StructuralFeatureValueActions Pulled-up fields must be non-private, if they are used somewhere in the subclass, because otherwise they would not be accessible anymore from the subclasses
Rename field	Changes the name of a field to a new name, and updates its references	No co-refactoring needed
encapsulate field	Creates getter and setter methods for the field and uses only those to access the field	Getter and setter activity have to be created (Readself & read/ AddStructuralFeatureValueAction) Replace StructuralFeatureValueActions to that field with CallOperationActions to the getter and setter, respectively

Figure 1 illustrates some quality metrics related to an activity diagram. We have one class *Circle* containing three properties and two operations. In Fig. 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 Fig. 1 for their formulas.

2.2 Refactorings

The refactoring of object-oriented programs is a well-researched domain (Fowler et al. 1999), and many of the identified refactorings for object-oriented programs have been adopted for the refactoring of design models (Sunye et al. 2001). 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



Classes	
# classes	= 2
PPC	= 1,5
OPC	= 1

Activity	
# edges	= 11
# nodes	= 7
# actions	= 4
CFC	= 4
LOC	= 0,4

Legend
PPC … # properties per class
OPC … # operations per class,
CFC … control-flow complexity = #forks + # decisions
LOC … locality = # reads in referenced class / all reads

Fig. 2 Motivating example—after extract class *point*

applying a refactoring to the class diagram in order to maintain the validity of consistency rules among classes and activities. Table 1 describes a complete list of the considered class diagram refactorings and the corresponding co-refactorings of activities:

2.2.1 *Consistency rules*

To avoid ambiguities regarding the semantics of classes, activities, and actions, we adopt the semantics of the Foundational Subset For Executable UML (fUML) (http://www.omg.org/spec/FUML/1.1/; Crane and Dingel 2008) 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.

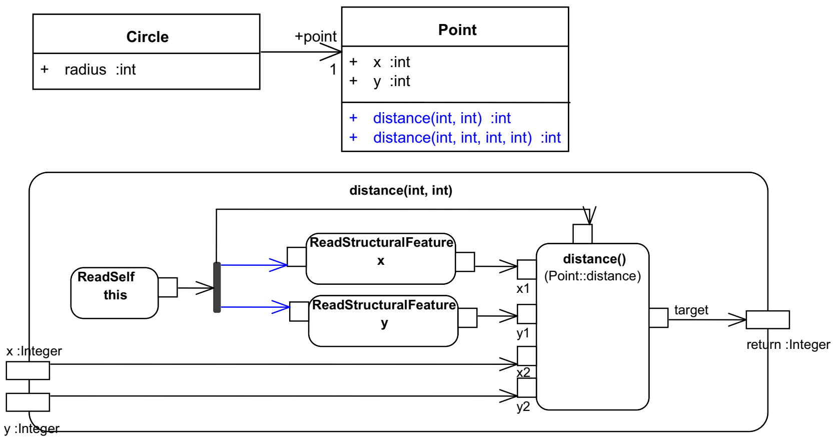
2.2.2 *Refactorings and co-refactorings*

We consider 15 *well-known refactorings of class diagrams* (Sunye et al. 2001; Boger et al. 2002) 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 Fig. 1.

As in our example, the class *Circle* contains two properties *x* and *y*, which specify the coordinates of its center point, and 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 are depicted in Fig. 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.



Classes	
# classes	= 2
PPC	= 1,5
OPC	= 1

Activity	
# edges	= 9
# nodes	= 5
# actions	= 4
CFC	= 3
LOC	= 1

Legend
PPC … # properties per class
OPC … # operations per class,
CFC … control-flow complexity = #forks + # decisions
LOC … locality = # reads in referenced class / all reads

Fig. 3 Motivating example—after move methods *distance*

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.

To improve this situation, we have to apply another refactoring, namely "Move Operation," in order to move the operation 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 *distance*(*int, int*), the operation *distance*(*int, int, int, int*) is called, which is not possible in the scope of *Point*, since *Point* has no access to the instance of *Circle*. Nevertheless, when we accept the temporary inconsistency and also move the operation *distance*(*int, int, int, int*) into the class *Point*, we obtain a new result, depicted in Fig. 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.

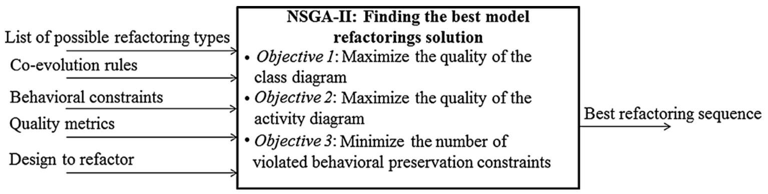
In conclusion, applying refactorings on the class diagram may have a strong impact on the quality of the activity diagrams that specify the behavior of the classes' operations. Even worse, in several scenarios, the class refactorings will break their consistency. Finding a good sequence of refactorings 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.

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 subsequently provide 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 improves the quality of different diagrams at the same time while preserving behavioral 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 violating 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 whether the behavior is changed after applying the refactorings on the class diagram.



List of possible refactoring types
Co-evolution rules
Behavioral constraints
Quality metrics
Design to refactor
NSGA-II: Finding the best model refactorings solution
• Objective 1: Maximize the quality of the class diagram
• Objective 2: Maximize the quality of the activity diagram
• Objective 3: Minimize the number of violated behavioral preservation constraints
Best refactoring sequence

Fig. 4 Multi-objective model refactoring: overview

The general structure of our approach is sketched in Fig. 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 also maximizes 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. (2002) to our problem domain (Fig. 5).

3.2 Multi-objective formulation

3.2.1 Nsga-II

The basic idea of NSGA-II (Deb et al. 2002) is to make a population of candidate solutions evolve toward the near-optimal solution in order to solve a multi-objective optimization problem. NSGA-II is designed to find a set of near-optimal solutions, called non-dominated solutions or the Pareto front. A non-dominated solution is one that provides a suitable compromise between all objectives without degrading any of them. As described in Algorithm 1, the first step in NSGA-II is to create randomly a population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P_0 using genetic operators such as crossover and mutation (line 2). Both populations are merged into a new population R_0 of size N (line 5).

Fast-non-dominated-sort is the algorithm used by NSGA-II to classify individual solutions into different dominance levels. Indeed, the concept of Pareto dominance consists of comparing each solution x with every other solution in the population until it is dominated by one of them. If no solution dominates it, the solution x will be considered non-

Fig. 5 High-level pseudo-code of NSGA-II

```
1.  Create an initial population $P_0$
2.  Generate an offspring population $Q_0$
3.  $t$=0;
4.  **while** *stopping criteria not reached* **do**
5.      $R_t = P_t \cup Q_t$;
6.      $F$ = fast-non-dominated-sort ($R_t$);
7.      $P_{t+1} = \emptyset$ and $i$=1;
8.      **while** $| P_{t+1}| + |F_i| \bullet N$ **do**
9.          Apply crowding-distance assignment($F_i$);
10.         $P_{t+1} = P_{t+1} \cup F_i$ ;
11.         $i = i+1$;
12.     **end**
13.     $Sort(F_i, \prec n)$;
14.     $P_{t+1} = P_{t+1} \cup F_i[1 : (N-| P_{t+1} |)]$;
15.     $Q_{t+1}$ = create-new-pop($P_{t+1}$);
16.     $t = t+1$;
17. **end**
```

dominated and will be selected by the NSGA-II to be a member of the Pareto front. If we consider a set of objectives f_i, $i,j \in 1\ldots n$, to maximize, a solution x dominates x'

$$\text{iff}\,\forall i,\, f_i(x')f_i(x)\,\text{and}\,\exists j|f_i(x') < f_i(x).$$

The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front F_0 get assigned dominance level of 0. Then, after taking these solutions out, fast-non-dominated-sort calculates the Pareto-front F_1 of the remaining population; solutions on this second front get assigned dominance level of 1, and so on. The dominance level becomes the basis of selection of individual solutions for the next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When NSGA-II has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance to make the selection (line 9). This parameter is used to promote diversity within the population. This front F_i to be split, is sorted in descending order (line 13), and the first ($N - |P_{t+1}|$) elements of F_i are chosen (line 14). Then, a new population Q_{t+1} is created using selection, crossover, and mutation (line 15). This process will be repeated until reaching the last iteration according to the stop criteria (line 4). It is interesting to mention that NSGA-II is an elitist algorithm that does not use any explicit archive of elite individuals. In fact, elitism is ensured by the crowded comparison operator that prefers solutions having better Pareto ranks. In this way, NSGA-II preserves elite solutions by keeping best non-dominated fronts in the race, and when considering the last non-dominated front, only least crowded solutions are selected from this latter to build the next population of N individuals.

3.2.2 NSGA-II adaptation for model refactoring

This section describes how NSGA-II (Deb et al. 2002) 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.

3.2.2.1 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. The size of the solution represents the number of refactoring (dimensions) in the vector. 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. For example, the controlling

Fig. 6 Representation of an NSGA-II individual

move method (*Circle, Point, distance(int, int)*)
move field (*Point, Circle, x*)
extract class(*Circle, Proprieties, radius, distance (int, int, int)*)

parameters of a move method refactoring are the source and target classes, and the method to move from the source class as described in the example of Fig. 6. Thus, the refactorings and its parameters are encoded as logic predicates (Strings). Each dimension of the vector is a logic predicate describing the refactoring type and its parameters.

An example of a solution is given in Fig. 6 on the class diagram of the motivating example of Fig. 2. This solution contains three dimensions that correspond to three refactorings applied to different parts of the source class diagram. For instance, the predicate *move method* (*Circle, Point, distance*(*int, int*)) means that the method *distance*(*int, int*) is moved from class *Circle* (source class) to class *Point* (target class).

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 Sect. 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 Sect. 2. For example, to apply the refactoring operation *move method* (*Circle, Point, distance*()), 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 exist; *distance*() declaration should be in the class *Point*; and *distance*() declaration should not exist in the class *Circle*. Other constraints checked by the activity diagram are discussed in Sect. 2.

3.2.2.2 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 (Bansiya and Davis 2002) described in Table 2. All the 11 metrics are aggregated in one fitness function with equal importance and normalized between 0 and 1.

Quality attribute	Definition Computation
Reusability	A design with low coupling and high cohesion is easily reused by other designs. $0.25 \times$ Coupling $+ 0.25 \times$ Cohesion $+ 0.5 \times$ Messaging $+ 0.5 \times$ Design size
Flexibility	The degree of allowance of changes in the design $0.25 \times$ Encapsulation $- 0.25 \times$ Coupling $+ 0.5 \times$ Composition $+ 0.5 \times$ Polymorphism

Quality attribute	Definition Computation
Understandability	The degree of understanding and the easiness of learning the design implementation details. 0.33 × Abstraction + 0.33 × Encapsulation − 0.33 × Coupling + 0.33 × Cohesion − 0.33 × Polymorphism − 0.33 × Complexity − 0.33 × Design size
Functionality	Classes with given functions that are publically stated in interfaces to be used by others. 0.12 × Cohesion + 0.22 × Polymorphism + 0.22 × Messaging + 0.22 × Design Size + 0.22 × Hierarchies
Extendibility	Measurement of design's allowance to incorporate new functional requirements. 0.5 × Abstraction − 0.5 × Coupling + 0.5 × Inheritance + 0.5 × Polymorphism
Effectiveness	Design efficiency in fulfilling the required functionality. 0.2 × Abstarction + 0.2 × Encapsulation + 0.2 × Composition + 0.2 × Inheritance + 0.2 × Polymorphism

2. *Quality of the activity diagram fitness function* represents an aggregation (sum) of 12 metrics described in Table 3. 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 Sect. 2.

3.2.2.3 Selection To guide the selection process, NSGA-II uses a binary tournament selection based on dominance and crowding distance (Deb et al. 2002). 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.

3.2.2.4 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

Table 2 QMOOD metrics for design properties (Bansiya and Davis 2002)

Design property	Metric	Description
Design size	DSC	Design size in classes
Complexity	NOM	Number of methods
Coupling	DCC	Direct class coupling
Polymorphism	NOP	Number of polymorphic methods
Hierarchies	NOH	Number of hierarchies
Cohesion	CAM	Cohesion among methods in class
Abstraction	ANA	Average number of ancestors
Encapsulation	DAM	Data access metric
Composition	MOA	Measure of aggregation
Inheritance	MFA	Measure of functional abstraction
Messaging	CIS	Class interface size

Table 3 Activity diagrams metrics

Metric	Description
NP	Number of parameters
NNO	Number of nodes
NED	Number of edges
NAC	Number of actions
CFC	Control-flow Complexity
ICOM	Interface complexity
HAC	Halstead-based activity complexity
CNC	Coefficient of network complexity
FIFO	Fan-in/fan-out metrics for activities
TD	Tree depth metric
TW	Tree width metric
LO	Locality

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. Each solution has a length limit in terms of number of refactorings. When applying the crossover operator, the new solution may have higher number of refactorings than the length limit (input of the algorithm). Thus, the algorithm randomly eliminates some of the dimensions of the vector (refactorings) to respect the size constraint. As illustrated in Fig. 7a, 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 with other ones from the initial list of possible refactorings. An example is shown in Fig. 7b. 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 after applying the change operators 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 modeling 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 modeling 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 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.

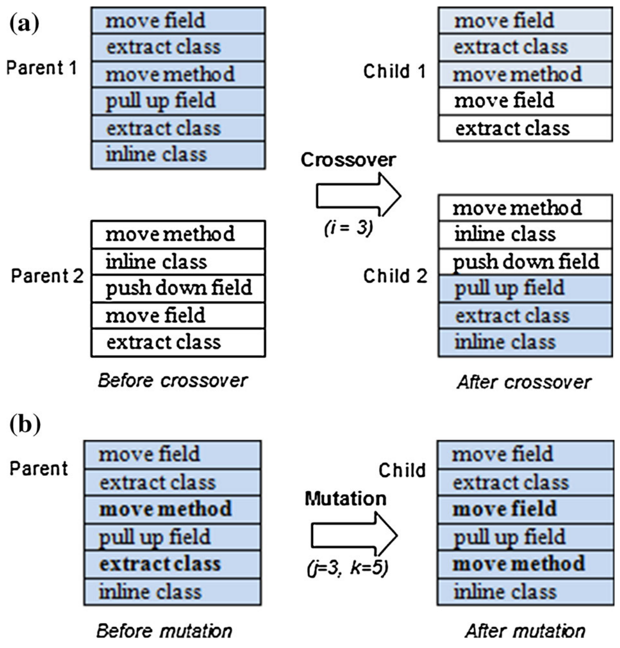


Fig. 7 Changes operators. **a** Crossover operator. **b** Mutation operator

4.1 Research questions

In our study, we assess the performance of our model refactoring approach of 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. To this end, we defined the following research questions:

RQ1 To what extent can the proposed approach improve 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 performs compared to an existing model refactoring approach (Moghadam and Cinneide 2012) 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 (Kessentini et al. 2011) 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,* Number of Fixed Defects, which 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.

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

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:

$$G - \text{Class Diagram} = \frac{\sum_{i=1}^{i=12} |q'_i - q_i|}{12} \text{ and } G - \text{Activity Diagram} = \frac{\sum_{i=1}^{i=11} |q'_i - q_i|}{11},$$

where $q_i{}'$ and q_i represent the value of the quality attribute i, respectively, after and before refactoring. As described in the previous section, we considered a total of 12 metrics related to class diagrams and 11 metrics for 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. The users evaluated the entire best solutions proposed by our approach. We define the metric "refactoring precision" (RP) which corresponds to the number of meaningful refactorings over the total number of suggested refactoring operations: $\text{RP} = \frac{\#\text{feasiblerefactorings}}{\#\text{proposed refactorings}}$

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). The use of a single-objective algorithm is to show that the two objectives of our multi-objective formulation are conflicting. If the two objectives were not conflicting then the results of NSGA-II will be similar to GA. Thus, in that case we will not need to propose a multi-view approach.

To answer *RQ4*, we compared our design refactoring results with a recent tool, called DesignImpl proposed recently by Iman and Mel (Moghadam and Cinneide 2012) that does not use heuristic search techniques. The current version of DesignImpl 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): $MP = \frac{|R| \cap |R_m|}{|R_m|}$, where R is the set of refactorings suggested by our tool and R_{m} is the set of refactorings suggested manually by software engineers. In fact, several equivalent refactoring solutions can be proposed to improve the quality. Thus, the tool can propose some refactorings that are different than those proposed by the designers, but improve the overall quality of the design. Thus, MP corresponds to the portion of the correct refactorings after manually evaluating them by the developers (that can be dissimilar from their suggestions).

4.2 Experimental settings

Our study considers 27 model fragments extracted from four open-source projects using the IBM Rational Rose tool (http://www-03.ibm.com/software/products/en/ratirosefami): Xerces-J, GanttProject (Gantt for short), JFreeChart, and Rhino. 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 4 summarizes for each model the number of detected design defects using our previous work (Kessentini et al. 2011), as well as the number of model elements. A model fragment is a set of model elements extracted from the open-source system. In fact, we extracted these model fragments from the different open-source systems (we did not consider the open-source system as one model, but we extracted several fragments from these systems). The number of elements in Table 4 is not the number of model fragments, but the number of elements in all the model fragments per open-source system.

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 six subjects from the University of Michigan, and some of them are working in automotive industry companies. Subjects include four master students in Software Engineering and two PhD students in Software Engineering. Four 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. The subjects manually evaluated the best refactoring solutions proposed by the different techniques. In addition, they manually refactor some of the detected design defects on the class diagrams. This outcome was compared with the solutions proposed by our techniques.

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 100,000 evaluations of 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

Table 4 The systems studied

Systems	Release	#Elements	#Smells
JFreeChart	v1.0.9	81	22
GanttProject	v1.10.2	114	28
Xerces-J	V2.7.0	96	31
Rhino	v1.7R1	88	26

probability of gene modification is 0.3; stopping criterion = 100,000 evaluations. The elitism can cause premature convergence since population members would be converging toward the same region of the search space. Based on our experimentations, we concluded that for our problem, it is effective and efficient to use an elitist schema while using a high mutation rate (0.8). The latter allows the continued diversification of the population, which discourage premature convergence to occur.

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 the 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 α (≤ 0.01) means that we accept H_1 and we reject H_0. However, a *p* value that is strictly greater than α (>0.01) means the opposite. 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.

The Wilcoxon signed-rank test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference in magnitude. The effect size could be computed by using the Cohen's *d* statistic (Cohen 1988). The effect size is considered: (1) small if $0.2 \leq d < 0.5$; (2) medium if $0.5 \leq d < 0{:}8$, or (3) large if $d > 0.8$. Table 5 gives the effect sizes in addition to the *p* values of the Wilcoxon test when comparing the results of NSGA-II to the GA.

We note that the mono-objective algorithm provides only one refactorings 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 (Rachmawati and Srinivasan 2009). 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 (Rachmawati and Srinivasan 2009) 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.

5 Results

5.1 Results for RQ1

As described in Fig. 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

Table 5 Statistical test results when comparing NSGA-II to the mono-objective approach

Scenario	JFreeChart	GanttProject	Xerces-J	Rhino
p value	6.12E-06	3.51E-09	8.27E-04	2.32E-04
Effect size	0.13	0.69	0.52	0.82

(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. The similar observation is also valid for the other techniques used in our experiments. This type of defect usually requires a large number of refactoring operations and is known to be very difficult to fix (Dag 2013).

Another observation is that our technique may introduce some new defects after refactoring. These new defects can be fixed by our approach since the fitness function counts the number of remaining defects in the system (to minimize) after executing the refactorings sequence.

In Figs. 9 and 10, we show the obtained gain values 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. As a consequence, we noticed that the quality of both class diagrams and activity diagrams is improved. The highest quality improvement scores of 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. Figure 10 shows that all the quality metrics were improved on all the systems except the functionality attribute for JFreeChart and Xerces-J. We looked to experiments data to understand the reason of the loss on Functionality of JFreeChart and Xerces-J. In fact, the functionality measure is calculated as the following: $0.12 \times$ Cohesion $+ 0.22 +$ Polymorphism $+ 0.22 \times$ Messaging $+ 0.22 \times$ Design size $+ 0.22 \times$ Hierarchies. We found that the design size of some JFreeChart and Xerces-J models after refactoring was lower than the design size before refactoring. Several move methods/fields were applied, leading to some empty classes after refactoring (thus not considered in the design size anymore). Furthermore, we found that the best refactoring solution included few extract class refactorings thus the design size was not increased with new classes (model

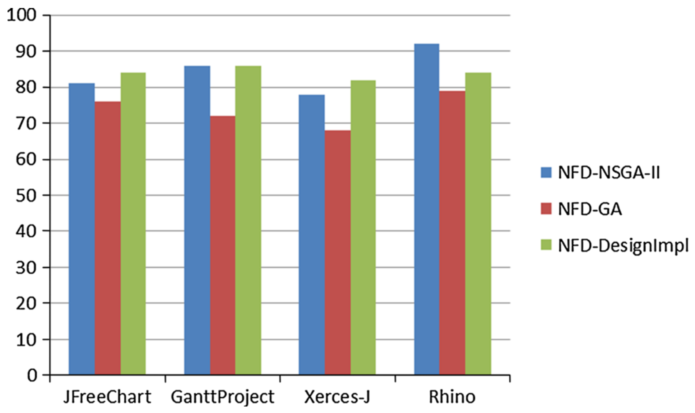


Fig. 8 NFD median values of NSGA-II, GA, and DesignImpl over 51 independent simulation runs using the Wilcoxon rank-sum test with a 99 % confidence level ($\alpha < 1$ %)

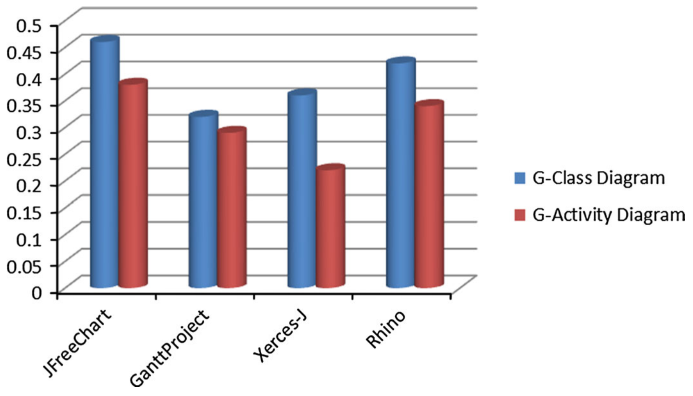


Fig. 9 Design quality improvements median values for class diagrams and activity diagrams of NSGA-II, GA, and DesignImpl over 51 independent simulation runs

elements). This can be explained by the fact that JFreeChart and Xerces-J were the only systems that did not include several large classes, and most of the classes have a small or medium size in terms of number of methods. Of course, the overall functionalities of the system were the same before refactoring as demonstrated later in RQ2 by the manual refactoring evaluation (RP).

5.1.1 Results for RQ2

As described in Fig. 11, we found that an average of more than 80 % of proposed refactoring operations are considered as feasible and do not generate behavior incoherence. A slight loss in the *NFD* and *G* is largely compensated by the significant improvement in terms of refactorings feasibility and behavior preservation.

5.1.2 Results for RQ3

As described in Figs. 8, 9, and 11, 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 value 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.

5.1.3 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 12 shows that most of the suggested refactorings by NSGA-II are similar to those applied by developers with an average of more than 70 %. In fact, we calculated the intersection between the recommended refactorings by NSGA-II and the manually suggested refactorings by the subjects over the total number of

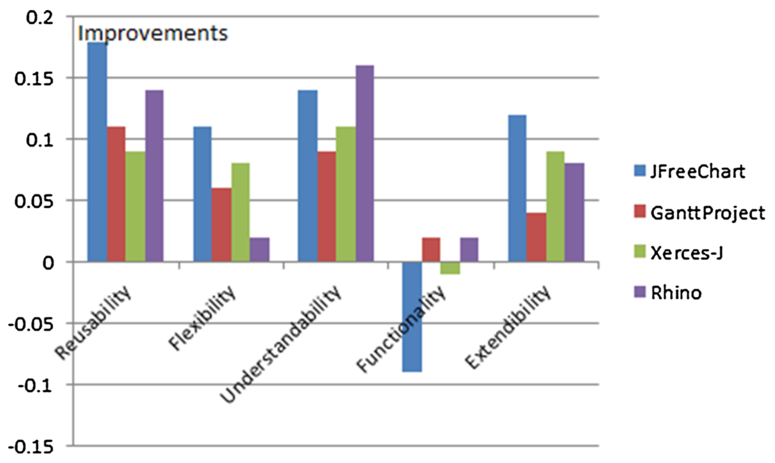


Fig. 10 Quality factors median values of NSGA-II, over 51 independent simulation runs

recommend refactorigs. 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 % confirms 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.

Another advantage related to the use of our multi-objective approach is the diversity of non-dominated solutions as described in Fig. 13. Figure 13 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 select the best trade-off solution between the three objectives based on their own preferences.

The selection of the "best" solution is based on the preferences of the developer. In fact, the developer may select a solution providing a high quality of class and activity diagrams, but violating several constraints since he has enough time to fix them before the next release for example. In another situation, the developer may select a solution that do not violate constraints (or only violating few of them) and slightly increase the quality of the models because he do not have enough time to fix the errors created by the constraints or if he want to minimize the risk related to the new refactorings. In case that the developer do not have any specific preferences and he wants to optimize all the three objectives at the same time, then he can select the solution at the knee point or the closest solution to the ideal point (the ideal point is (0,0,0) if all the objectives are to minimize and normalized between zero and one).

Based on the plots of Fig. 13, 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 (Fowler et al. 1999) 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.

It is important to contrast the results of multiple executions with the execution time to evaluate the performance and the stability of our approach. In fact, usually in the optimization research field, the most time-consuming operation is the evaluation step. All the

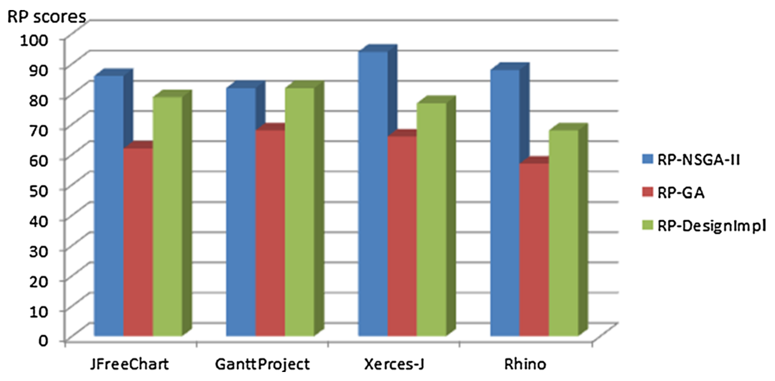


Fig. 11 The refactoring precision (RP) median values of NSGA-II, GA, and DesignImpl over 51 independent simulation runs using the Wilcoxon rank-sum test with a 99 % confidence level ($\alpha < 1$ %)

algorithms under comparison were executed on machines with Intel Xeon 3 GHz processors and 4 GB RAM. The execution time for finding the optimal refactoring solution with 100,000 evaluations was 48 min as an average execution time of all case studies (models). The average time required by the developers to fix the defects in the different systems was more than two hours. The execution of our approach can be acceptable since it is executed, in general, up front (at night) to find suitable refactorings.

5.2 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 three 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. For the selection threat, the subject diversity in terms of profile and experience could affect our study. First, all subjects were volunteers. We also mitigated the selection threat by giving written guidelines and examples of refactorings already evaluated with arguments and justification. Additionally, each group of subjects evaluated different operations from different systems using different techniques/algorithms.

We take into consideration the internal threats to validity in the use of stochastic algorithms since 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 ($\alpha = 1$ %). However, the parameter tuning of the different optimization algorithms used in our experiments creates

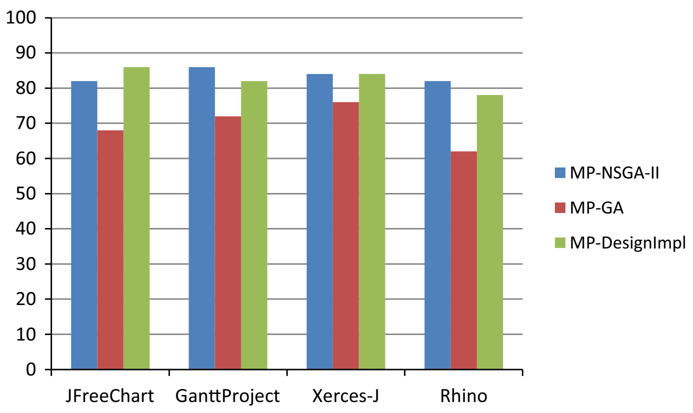


Fig. 12 The MP median values of NSGA-II, GA, and DesignImpl over 51 independent simulation runs using the Wilcoxon rank-sum test with a 99 % confidence level ($\alpha < 1$ %)

another internal threat that we need to evaluate in our future work by additional experiments. 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. In fact, parameter tuning of search algorithms is still an open research challenge till today. We have used the trial-and-error method which is one of the most used ones. However, the use of ANOVA-based technique could be another interesting direction from the viewpoint of the sensitivity to the parameter values.

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 and evaluate the scalability of our approach with larger models.

6 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.

6.1 Model refactorings

Two surveys concerning model refactorings are available (Mohamed et al. 2009; Mens et al. 2007a), proposed by Mens et al., that discuss different research trends and classifications for model refactoring. One of the first investigations in this area was done by Sunyè et al. (2001) who define a set of UML refactorings on the conceptual level by expressing pre- and post-conditions in OCL. Boger et al. (2002) present a refactoring browser for UML supporting the automatic execution of pre-defined UML refactorings. While these

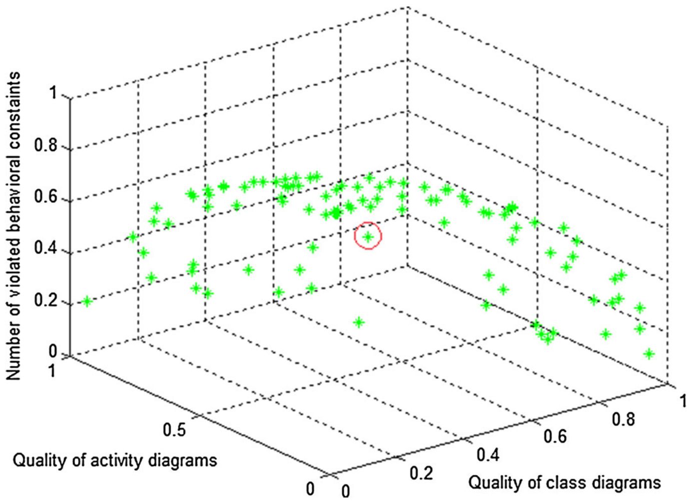


Fig. 13 Pareto front for NSGA-II obtained on Xerces-J

two approaches focus on pre-defined refactorings, approaches by Porres (2005), Zhang et al. (2005), and Kolovos et al. (2007) discuss the introduction of user-defined refactorings by using dedicated textual languages for their implementation. A similar idea is followed in (Biermann et al. 2006; Mens 2006) Biermann et al., 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 (France et al. 2003) Sun et al.

The mentioned approaches cover mostly single-view refactorings and focus on the implementation of semi-automatic executable refactorings. Only some approaches for tackling consistency between different views in the context of refactorings have been presented. For instance, Markovic and Baar (2008), and Correa and UML (2004) proposed to refactor UML class diagrams, also adapting attached OCL constraints. Another approach that considers the effect of refactorings of UML class diagrams on operations implemented in OCL with respect to behavioral equivalence is presented in Sun et al. (2013). A constraint-based refactoring approach for UML is presented in (Steimann 2011), proposed by Steimann, 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 Reimann et al. (2010) by specifying a generic role-based refactorings that can be bound to specific languages. Another approach aiming for generic model refactorings is presented in Wimmer et al. (2012) 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 Arendt and Taentzer (2013), a 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 Arcelli et al. (2012). 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 (Eramo et al. 2008). We have early works on multi-view consistency (Kolovos et al. 2007; Mens 2006) 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 (Van Der Straeten et al. 2004; Van Gorp et al. 2003) and in particular bidirectional model transformations (Moha et al. 2009; France et al. 2003). Other approaches use so-called correspondence rules for synchronizing models in the contexts of RM-ODP and model-driven web engineering (Cicchetti et al. 2009; Grundy et al. 1998; Wimmer et al. 2012). All these approaches have in common that they consider only atomic changes when reconciling models and not refactorings. In previous work (Wimmer et al. 2012), we presented coupled transformations to refactor different views altogether by automatically executing 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 (Ráth et al. 2009) based on a kind of event/condition/action rules.

In previous work (Ghannem et al. 2013), 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.

6.2 Model/code refactorings

The synchronization of models and code is of course also a challenging issue when it comes to refactoring. In Bottoni et al. (2003), distributed graph transformations are used to specify coupled refactorings on UML models and Java code. Van Gorp et al. (2003) have presented an extension of the UML metamodel which allows expressing the pre- and post-conditions 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 (MoDELS 2011) discussed before has been also extended to dealing model/code co-refactoring by so-called bridge constraints that capture the correspondences between model elements and the code elements (von Pilgrim et al. 2013).

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 the code as a multi-objective optimization problem.

6.3 Code refactorings

Harman et al. (2007) 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 a 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 preformed improvements. Recently, Ó Cinnéide et al. (2012) 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.

7 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 with some of those discussed in the literature (Moghadam and Cinneide 2012). 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 (Moghadam and Cinneide 2012).

In future work, we are planning to perform 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

Arcelli, D., Cortellessa, V., & Trubiani, C. (2012). Antipattern-based model refactoring for software performance improvement. In *QoSA*, pp. 33–42.

Arendt, T., & Taentzer, G. (2013). A tool environment for quality assurance based on the Eclipse modeling framework. *Automated Software Engineering, 20*(2), 141–184.

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

Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., & Weiss, E. (2006). Graphical definition of in-place transformations in the Eclipse modeling framework. In *MoDELS'06*. LNCS (Vol. 4199, pp. 425–439). Springer.

Bock, Conrad. (2003). UML 2 activity and action models, Part 2: Actions. *Journal of Object Technology, 2*(5), 41–56.

Boger, M., Sturm, T., & Fragemann, P. (2002). Refactoring browser for UML. In *NetObjectDays'02*. LNCS (Vol. 2591, pp. 366–377). Springer.

Bottoni, P., Parisi-Presicce, F., & Taentzer, G. (2003). Specifying integrated refactoring with distributed graph transformations. In *AGTIVE 2003*, pp. 220–235.

Cardoso, J., Mendling, J., Neumann, G., & Reijers, H. A. (2006). A discourse on complexity of process models. In *BPM Workshops*.

Cicchetti, A., Ruscio, D. D., & Pierantonio, A. (2009). Managing dependent changes in coupled evolution. In *ICMT'09*. LNCS (Vol. 5563, pp. 35–51). Springer.

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences*. Mahwah: Lawrence Erlbaum Associates.

Correa, A., & Werner, C. (2004). Applying refactoring techniques to UML/OCL models. In *Proceedings of Int'l Conference UML 2004*. LNCS (Vol. 3273, pp. 173–187). Springer.

Crane, M. L., & Dingel, J. (2008). Towards a formal account of a foundational subset for executable UML models. In *Model driven engineering languages and systems* (pp. 675–689). BerlIn Springer.

Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation, 6*, 182–197.

Eramo, R., Pierantonio, A., Romero, J. R., & Vallecillo, A. (2008). Change management in multiviewpoint systems using ASP. In *WODPEC'08*. IEEE.

Fenton, N., & Pfleeger, S. L. (1997). *Software metrics: A rigorous and practical approach*. London, UK: International Thomson Computer Press.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the design of existing code*. Boston, MA: Addison-Wesley. ISBN: 0-201-48567-2.

France, R. B., Ghosh, S., Song, E., & Kim, D.-K. (2003). A metamodeling approach to pattern-based model refactoring. *IEEE Software, 20*(5), 52–58.

Ghannem, A., El Boussaidi, G., & Kessentini, M. (2013). Model refactoring using interactive genetic algorithm. In G. Ruhe & Y. Zhang (Eds.), *Search based software engineering* (pp. 96–110). Berlin, Heidelberg: Springer.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization and machine learning*. Boston: Addison-Wesley Longman Publishing Co., Inc.

Grundy, J., Hosking, J., & Mugridge, W. B. (1998). Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering, 24*(11), 960–981.

Harman, M., & Tratt, L. (2007). Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual conference on genetic and evolutionary computation* (pp. 1106–1113). ACM, 2007.

http://www-personal.umd.umich.edu/~marouane/sqj15.htm.

http://www.omg.org/spec/FUML/1.1/.

http://www-03.ibm.com/software/products/en/ratirosefami.

Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., & Ouni, A. (2011). Design defects detection and correction by example, In *19th IEEE ICPC11* (pp. 81–90), Kingston, Canada.

Kolovos, D. S., Paige, R. F., Polack, F., & Rose, L. M. (2007). Update transformations in the small with the Epsilon wizard language. *JOT, 6*(9), 53–69.

Ma, H., Shao, W., Zhang, L., Ma, Z., & Jiang, Y. (2004). Applying OO metrics to assess UML meta-models. In *«UML» 2004—The Unified Modeling Language. Modeling Languages and Applications* (pp. 12–26). Berlin, Heidelberg: Springer.

Markovic, S., & Baar, T. (2008). Refactoring OCL annotated UML class diagrams. *Software and Systems Modeling, 7*(1), 25–47.

Mens, T. (2006). On the use of graph transformations for model refactoring. In *Generative and transformational techniques in software engineering*. LNCS (Vol. 4143, pp. 219–257). Springer.

Mens, T., Taentzer, G., & Müller, D. (2007a). Challenges in model refactoring. In *Proceedings of 1st workshop on refactoring tools*. University of Berlin.

Mens, T., Taentzer, G., & Runge, O. (2007b). Analyzing refactoring dependencies using graph transformation. *Journal on Software and Systems Modeling, 6*, 269.

Mens, T., & Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering, 30*(2), 126–139.

Moghadam, I. H., & Cinneide, M. O. (2012). Automated refactoring using design differencing. In *Software maintenance and reengineering (CSMR), 2012 16th European conference on* (pp. 43–52). IEEE.

Moha, N., Mahé, V., Barais, O., & Jézéquel, J. M. (2009). Generic model refactorings. In *Model driven engineering languages and systems* (pp. 628–643). Berlin, Heidelberg: Springer.

Mohamed, M., Romdhani, M., & Ghédira, K. (2009). Classification of model refactoring approaches. *JOT, 8*(6), 143–158.

Ó Cinnéide, M., Tratt, L., Harman, M., Counsell, S., & Hemati Moghadam, I. (2012, September). Experimental assessment of software metrics using automated refactoring. In *Proceedings of the ACM-IEEE international symposium on empirical software engineering and measurement* (pp. 49–58). ACM.

Porres, I. (2005). Rule-based update transformations and their application to model refactorings. *Software and Systems Modeling, 4*(4), 368–385.

Rachmawati, L., & Srinivasan, D. (2009). Multiobjective evolutionary algorithm with controllable focus on the knees of the pareto front. *IEEE Transactions on Evolutionary Computation, 13*(4), 810–824.

Ráth, I., Varró, G., & Varró, D. (2009). Change-driven model transformations. In *MODELS'09*. LNCS (Vol. 5795, pp. 342–356). Springer.

Reimann, J., Seifert, M., & Aßmann, U. (2010). Role-based generic model refactoring. In *Model driven engineering languages and systems* (pp. 78–92). Berlin, Heidelberg: Springer.

Sjøberg, D. I. K., Yamashita, A. F., Anda, B. C. D., Mockus, A., & Dybå, T. (2013). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering, 39*(8), 1144–1156.

Steimann, F. (2011). Constraint-based model refactoring. In *Model driven engineering languages and systems* (pp. 440–454). Berlin, Heidelberg: Springer

Sun, W., France, R. B., & Ray, I. (2013). Analyzing behavioral refactoring of class models. In *ME@MoDELS 2013*, pp. 70–79.

Sunye, G., et al. (2001). Refactoring UML models. In *Proceedings of UML.*

Sunyé, G., Pollet, D., Traon, Y. L., & Jézéquel, J. M. (2001). Refactoring UML models. In *UML'01. LNCS*, Vol. 2185 (pp. 134–148). Springer.

Van Der Straeten, R., Jonckers, V., & Mens, T. (2004). Supporting model refactorings through behaviour inheritance consistencies, In *UML. LNCS*, Vol. 3273 (pp. 305–319), Springer.

Van Gorp, P., Stenten, H., Mens, T., & Demeyer, S. (2003). Towards automating source-consistent UML refactorings, In *UML*. LNCS, Vol. 2863 (pp. 144–158). Heidelberg: Springer.

Van Kempen, M., Chaudron, M., Koudrie, D., & Boake, A. (2005). Towards proving preservation of behaviour of refactoring of UML models. In *Proceedings of SAICSIT 2005,* pp. 111–118.

von Pilgrim, J., Ulke, B., Thies, A., & Steimann, F. (2013). Model/code co-refactoring: An MDE approach. In *ASE*, pp. 682–687.

Wimmer, M., Moreno, N., & Vallecillo, A. (2012). Viewpoint co-evolution through coarse-grained changes and coupled transformations. *TOOLS, 50*, 336–352.

Zhang, J., Lin, Y., & Gray, J. (2005). Generic and domain-specific model refactoring using a model transformation engine. In *Model-driven software development—research and practice in software engineering* (pp. 199–217). Springer.

Usman Mansoor is currently a Ph.D. student in computer science at the University of Michigan under the supervision of Pr. Marouane Kessentini (University of Michigan). He is a member of the SBSE@Michigan research laboratory, University of Michigan, USA. Previously he was a Graduate Research Student of Computer Engineering in Ajou University, South Korea under Brain Korea (BK21) scholarship initiative undertaken by Korean Government. His research interests include the application of artificial intelligence techniques to software engineering (search-based software engineering), software refactoring, software quality and model-driven engineering.

Marouane Kessentini is a tenure-track assistant professor at University of Michigan. He is the founder of the research group: Search-based Software Engineering@Michigan. He holds a Ph.D. in Computer Science, University of Montreal (Canada), 2011. His research interests include the application of artificial intelligence techniques to software engineering (search-based software engineering), software testing, model-driven engineering, software quality, and re-engineering. He has published around 50 papers in conferences, workshops, books, and journals including three best paper awards. He has served as program-committee/organization member in several conferences and journals.

Manuel Wimmer is postdoctoral researcher in the Business Informatics Group (BIG) at the Vienna University of Technology, Austria where he received his Ph.D. in 2008. He has been a research associate in the Software Engineering Group at the University of Málaga in 2011/2012. His current research interests comprise Web engineering, model engineering, and ontology engineering. He is/was involved in several national and international projects dealing with the application of model engineering techniques for domains such as tool interoperability, versioning, social Web, and Cloud computing. He is coauthor of the book Model-driven Software Engineering in Practice (Morgan & Claypool, 2012) and coauthor of more than 120 scientific articles published in international conferences (e.g., ICWE, MODELS, ASE, ICMT) and journals (e.g., ACM CSUR, SoSym, JSS, JOT). Furthermore, he has served as workshop co-chair for ICWE in 2012. For a more detailed curriculum vitae and list of publications, please visit http://www.big.tuwien.ac.at/staff/mwimmer.

Kalyanmoy Deb is Koenig Endowed Chair Professor at Electrical and Computer Engineering in Michigan State University, USA. Prof. Deb's research interests are in evolutionary optimization and their application in optimization, modeling, and machine learning. Prof. Deb has numerous awards and honours in his name, including the prestigeous Shanti Swarup Bhatnagar Prize in Engineering Sciences in 2005, "Thomson Citation Laureate Award," an award given to an Indian Researcher for making most highly cited research contribution during 1996-2005 in a particular discipline according to ISI Web of Science., Friedrich Wilhelm Bessel Research Award and Humboldt Fellowship from Alexander von Humboldt Foundation, Germany. He is a fellow of Indian National Science Academy (INSA), Indian National Academy of Engineering (INAE), Indian Academy of Sciences (IASc), and International Society of Genetic and Evolutionary Computation (ISGEC). He has been awarded "Distinguished Alumnus Award" from his Alma mater IIT Kharagpur in 2011. Author of more than 275 research papers, two textbooks, 17 edited books, his 2001 book on Evolutionary Multiobjective Optimization Algorithms is the first ever compilation of multiobjective optimization algorithms. Because of his pioneering research in the field of evolutionary multi-objective optimization (EMO), he has been invited to present 35 Keynote lectures and more than 100 invited lectures and tutorials on the topic. His NSGA-II paper from IEEE Trans. on Evolutionary Computation (2000) is judged as the Fast-Breaking Paper in Engineering by ESI Web of Science and now this paper is awarded the "Current Classic" and "Most Highly Cited Paper" by Thomson Reuters. He is fellow of IEEE and three science academies in India. He has published 350+ research papers with Google Scholar citation of 55,000+ with h-index 77. He is in the editorial board on 20 major international journals. More information about his research can be found from http://www.egr.msu.edu/~kdeb.