Empir Software Eng (2017) 22:670–715
DOI 10.1007/s10664-016-9442-8

Search-based detection of model level changes

Marouane Kessentini[1] · **Usman Mansoor**[1] · **Manuel Wimmer**[2] · **Ali Ouni**[3] · **Kalyanmoy Deb**[4]

Published online: 22 September 2016

Abstract Software models, defined as code abstractions, are iteratively refined, restructured, and evolved due to many reasons such as reflecting changes in requirements or modifying a design to enhance existing features. For understanding the evolution of a model a-posteriori, change detection approaches have been proposed for models. The majority of existing approaches are successful to detect atomic changes. However, composite changes, such as refactorings, are difficult to detect due to several possible combinations of atomic changes or eventually hidden changes in intermediate model versions that may be no longer available. Moreover, a multitude of refactoring sequences may be used to describe the same model evolution. In this paper, we propose a multi-objective approach to detect model changes as a sequence of refactorings. Our approach takes as input an exhaustive list of possible types of model refactoring operations, the initial model, and the revised model, and generates as output a list of refactoring applications representing a good compromise between the following two objectives (i) maximize the similarity between the expected revised model and the generated model after applying the refactoring sequence on the initial model, and (ii) minimize the number of atomic changes used to describe the evolution. In fact, minimizing the number of atomic changes can important since it is maybe easier for a designer to understand and analyze a sequence of refactorings (composite model changes) rather than an equivalent large list of atomic changes (Weissgerber and Diehl 2006). Due to the huge number of possible refactoring sequences, a metaheuristic search method is used to explore the space of possible solutions. To this end, we use the non-dominated sorting genetic algorithm (NSGA-II) to find the best trade-off between our two objectives. The paper reports on the results of an empirical study of our

Communicated by: Andrea De Lucia

✉ Marouane Kessentini
marouane@umich.edu

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

[2] Vienna University of Technology, Wien, Austria

[3] Osaka University, Osaka, Japan

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

multi-objective model changes detection technique as applied on various versions of real-world models taken from open source projects and one industrial project. We compared our approach to the simple deterministic greedy algorithm, multi-objective particle swarm optimization (MOPSO), an existing mono-objective changes detection approach, and two model changes detection tools not based on computational search. The statistical test results provide evidence to support the claim that our proposal enables the generation of changes detection solutions with correctness higher than 85 %, in average, using a variety of real-world scenarios.

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

1 Introduction

Model-Driven Engineering (MDE) (Vermolen et al. 2011) considers models as first-class artifacts during the software lifecycle. The number of available tools, techniques, and approaches for MDE is increasing and more attention is paid to the evolution aspects in MDE (Küster et al. 2008; Kehrer et al. 2011; Gasevic et al. 2009; Mansoor et al. 2015a) along with the growing importance of modeling in software development. In fact, software models, defined as code abstractions, are iteratively refined, restructured, and evolved due to many reasons such as correcting errors in design; reflecting changes in requirements; and modifying a design to enhance existing features. Thus, effective techniques to evolve models as well as to understand their evolution are a must to help programmers integrating or modifying the features of an existing system.

One of the used techniques for evolving software systems is refactoring, which improves design structure while preserving external behavior (Fowler et al. 1999). Various tools (Moha et al. 2009; Liu et al. 2009; Harman and Tratt 2007; Seng et al. 2006; Mens and Tourwé 2004; Ben Fadhel et al. 2012) supporting refactoring have been proposed in the literature. The vast majority of these tools provides different environments to apply manually or automatically refactorings to avoid and fix bad-design practices (Brown et al. 1998). However, few approaches propose to detect changes between two (or more) model versions by composing atomic changes, such as adding, deleting, and updating model elements, to more coarse-grained changes such as refactorings.

The detection of changes between different versions of models can be beneficial for developers in different practical scenarios such as:

1. Collaborative conflict detection and resolution in model versioning and models merging: When models are changed in parallel, they have to be merged eventually to obtain a consolidated model. Therefore, several approaches have been proposed for detecting the operations that have been applied by developers on different local versions of the model (Wieland et al. 2013; Brosch et al. 2012; Mansoor et al. 2015b). Once the applied operations are available, conflict detection algorithms are used to identify pairs of operations that interfere with each other. In this regard, a conflict denotes a pair of operations, where one operation masks the effect of the other (i.e., they do not commute) or one operation disables the applicability of the other. An example for the former is a pair of parallel operations that update the same feature in the model with different values. The

latter case is at hand if one operation's preconditions are not valid anymore after applying the operations of the other developer. Such a scenario frequently occurs if composite operations (a sequence of cohesive atomic changes), e.g., model refactorings, are applied, because they may have potentially complex preconditions that may easily be invalidated by parallel operations. Thus, the detection of changes between different versions of a model is required to be then analyzed for the detection of conflicts when merging different versions of a model.

2. Regression testing on UML models: Several studies proposed techniques for the selection and generation of test cases for detected model changes between different versions of UML models (Briand et al. 2002; Sahin et al. 2015; Pilskalns et al. 2006). Thus, detected model changes are crucial for the regression testing phase.
3. Better understanding of evolved models: When integrating new features or updating existing features, it is maybe not enough for the developers to only look at the last version of the model. In fact, a developer has to understand what changes have been introduced by other developers to identify what are the model fragments to modify for the update of an existing requirement or integrating a new one (e.g. when a developer did not find a method that he introduced in previous releases).
4. Learning from detected model changes (history of changes): several studies proposed to use the history of detected changes to recommend new refactorings (Ouni et al. 2013; Porres 2003; Gîrba and Ducasse 2006 May 1).

In this regard, we distinguish between two categories of existing work: the first category (Dig et al. 2006; Weissgerber and Diehl 2006; Kim et al. 2012; Prete et al. 2010; Brosch et al. 2009) produces only atomic differences and the second category (Vermolen et al. 2011; Küster et al. 2008; Kehrer et al. 2011) is able to produce composed differences such as detecting refactorings. Our work can be classified in the second category. In general, existing approaches propose to detect differences between model versions using pre- and postconditions specified for each refactoring. In this case, the specified conditions are related to the possible changes that can be detected by comparing between source and revised models. However, some problems limit their effectiveness.

First, it is not possible to find the applied refactorings, if they have been performed in an overlapping sequence. The main reason is because their pre- and postconditions might not be valid due to preceding or succeeding refactorings when only considering the initial and the final versions of a model in the absence of each intermediate version after every single refactoring. Second, the list of possible changes and their combination may be very large for some modeling languages, e.g., consider UML where most refactorings of object-oriented programming may be applied on class diagrams. Thus, it is a fastidious task to specify detection rules for each possible refactoring combination, which is currently required by most of the existing approaches. Third, the evolution of a model from an initial version to a revised version can be described using different refactoring sequences having the same result. In fact, some complex refactorings are equivalent to a specific composition of basic refactorings. For instance, an *Extract Class* refactoring is a combination of a set of atomic operations such as create class, *Move Field*, *Move Method*, etc. Thus, some criteria have to be used to choose the best solution from equivalent ones. One of the important criteria, in addition to correctness of detected changes, is the number of refactorings used to describe the model evolution. In general, minimizing the number of refactorings is equivalent to maximizing the number of large composite refactorings used to describe the evolution. In fact, it is easier for a designer to

understand and analyze a sequence of refactorings (composite model changes) rather than a large list of atomic (simple) changes (Weissgerber and Diehl 2006). A mono-objective approach where the main goal is only to maximize correctness raises these questions: How to choose the best solution from a set of equivalent solutions in terms of correctness? How much correctness should be sacrificed for an improvement in understanding the detected changes?

In this paper, we propose to overcome the above-mentioned limitations. In fact, existing approaches are strict in the sense that the pre- and postconditions have to be fulfilled by detected refactorings on the initial and revised model, respectively. In this work, we have a relaxed notion. We only consider the hypothesis that the revised model and the computed revised model (produced by applying a sequence of changes to the initial model) should be equivalent and that the pre- and postconditions of the detected refactorings have to hold in the computed intermediate model versions when applying the refactorings, but we do not require that the pre- and postconditions hold in the initial and revised models, respectively. Furthermore, as developers understand easier evolutions described with a minimum set of refactorings we take into consideration the number of refactorings used to describe the evolution rather than the description of the detected changes using atomic operations. To this end, we propose to consider the detection of refactorings between model versions as a multi-objective optimization problem where every point (solution) in the search space is a new version of the initial model.

Our approach takes as input an exhaustive list of possible refactorings, the initial model and the revised one, and generates as output a list of detected changes in terms of refactorings. In this regard, a solution is defined as the combination of refactoring operations that should (i) maximize the similarity between the revised model and the generated model after applying the refactoring sequence on the initial model and (ii) minimize the number of refactorings used to describe the evolution. Due to the large number of possible solutions, a search-based method is used instead of an enumerative one to explore the space of possible solutions. We propose a multi-objective optimization approach to find the best compromise between these two conflicting objectives. The proposed algorithm is an adaptation of the Non-dominated Sorting Genetic Algorithm (NSGA-II) (Deb et al. 2002). NSGA-II aims at finding a set of representative Pareto optimal solutions in a single run. The evaluation of these solutions is based on the two aforementioned criteria.

Of course, the problem addressed in this paper is not related to detect regular refactorings between different releases that just improve the quality but also any changes (even those related to changing or adding features). In addition, we aim to describe the changes between different releases using composite refactorings and not atomic operations like most of existing work and tools such as Git or SVN. Version control tools such as SVN/CVS are more used at the code level rather than the model level. The advantage of our current approaches that it did not require the history of previous commit messages (not always available and accurate in real world settings, especially with old and legacy system projects) as input to generate detected refactorings.

We report experiments on seven large open source systems and one industrial project provided by our industrial partner, the Ford Motor Company, aimed at investigating to what extent model changes can be correctly detected using our multi-objective adaptation and to what extent the number of refactoring can be reduced to detect model changes. We also evaluate the performance of our detection technique comparing to another multi-objective algorithm and our previous work (Ben Fadhel et al. 2012) where only a mono-objective technique is used. In addition, we compared our work with two existing work not based on search techniques (Langer et al. 2012; Xing and Stroulia 2005).

The primary contributions of the paper can be summarized as follows:

- While there are attempts to investigate the detection of changes between different model versions, to the best of our knowledge (Harman et al. 2009) there is no work about the use of multi-objective optimization techniques for model changes detection. In fact, this work represents an extension of our previous ICSM2012 study that was the first search based approach for the detection of changes using a mono-objective approach (Ben Fadhel et al. 2012). Our proposal does not require an expert to write detection rules for each possible change combination. The rules for applying the changes are sufficient for their detection. Furthermore, we identify hidden (implicit) changes that are currently not detectable by existing approaches. In addition, we do not consider only correctness (similarity between given revised model and computed revised model) to evaluate the quality of proposed refactoring solutions but also minimizing the use of atomic changes to describe the refactoring solutions.
- We report the results of our statistical evaluation of our approach; we used large models extracted from seven open source systems and one industrial project that have an extensive evolution history. These models were analyzed manually to find the changes between different versions. We also report the comparison results of our approach with state of the art on model changes detection techniques.

The remainder of this paper is structured as follows. The next section gives an overview on the challenges of model operation detection by using a motivating example. Section 3 describes our adaptation of multi-objective optimization techniques to detect applied refactoring between different model versions. Section 4 describes obtained results of our experiment. We discussed the threats related to our experiments in Section 5. Section 6 summarizes related work. Finally, conclusions and future work are given in Section 6.

2 Background and Problem Statement

In this section, we first provide the necessary background of detecting model changes in MDE and discuss the importance of having the valuable information on high-level changes that have been applied between two versions of a model. Based on a motivating example, we subsequently illustrate the challenges addressed in this paper to reconstruct the list of applied operations from two versions of a model.

2.1 Background

MDE has been proposed as a new paradigm for raising the level of abstraction. In MDE, models are considered as central artifacts in the software engineering process, going beyond their traditional use as sketches and documentation. Instead, they act as single source of information for generating an executable software system. Thus, models are continuously evolved by developers during the development process of MDE-based projects.

2.1.1 Model Operations and Their Usage

The transition from code to models induces the need for adequate techniques to cope with the continuous and concurrent evolution of models. One of the most important tasks in this realm

is the detection and analysis of operations that have been applied between two versions of a model. In general, we may distinguish between two categories of operations. The first category concerns *atomic operations*, such as additions, deletions, updates, and moves. The second category comprises *composite operations* (Vermolen et al. 2011) consisting of a set of cohesive atomic operations, which are applied within one transaction to achieve one common goal. Such operations usually have well-defined pre- and postconditions that specify their applicability and their intended effect. The most prominent class of composite operations concerns refactorings introduced by Opdyke (Fowler et al. 1999). However, composite operations are not limited to refactorings; they may be used to implement any kind of in-place model transformation for supporting a specific purpose, such as model completion, refinement, and evolution.

As reported in (France and Rumpe 2007), the availability of the information on applied composite operations is a crucial prerequisite for automating several model management tasks. These tasks include model versioning and merging, the comprehension of a model's evolution, and the co-evolution of models, such as the migration of models to new metamodel versions and the synchronization of models and multiple views on them.

2.1.2 Operation Recording

The set of applied operations can be obtained by recording them directly in the modeling editor as they are applied by a developer (Küster et al. 2008). Although operation recording allows for obtaining a very precise change log efficiently, this technique is often not applicable in practice, because a tight integration with the modeling editor is necessary. Even if this tight integration is possible, users may still modify models using different editors or they may apply model transformation tools, which both hinders recording the applied operations.

Furthermore, users may apply a set of atomic operations manually having in combination the effect of a composite operation, which is happening frequently in practice (Vermolen et al. 2011); in such a scenario, operation recording approaches are not capable of recording the application of the composite operation, because the composite operation has not been triggered by the user explicitly. Thus, operation logs are often missing certain model evolution steps or are not available at all. In such cases, the operations have to be obtained by applying model comparison tools analyzing the initial and revised models.

2.1.3 Model Comparison

Several remarkable techniques have been proposed for deriving the applied operations a posteriori from two versions of a model. However, most of the existing model comparison tools are capable of detecting only atomic operations and, thus, do not provide the crucial information for enabling several model management tasks that also require the explicit knowledge on applied composite operations.

An example for atomic changes and how they are represented is shown in Fig. 1. In this figure, two versions of a simple UML class diagram representing different kinds of services are shown in the upper and lower half of the figure, whereas in the middle the atomic operations between the two versions are represented. Although, for this simple example the differences are not too complex to understand, larger examples that involve a higher number of operations are challenging to be understood. Furthermore, for reasoning on the evolution of models, a set

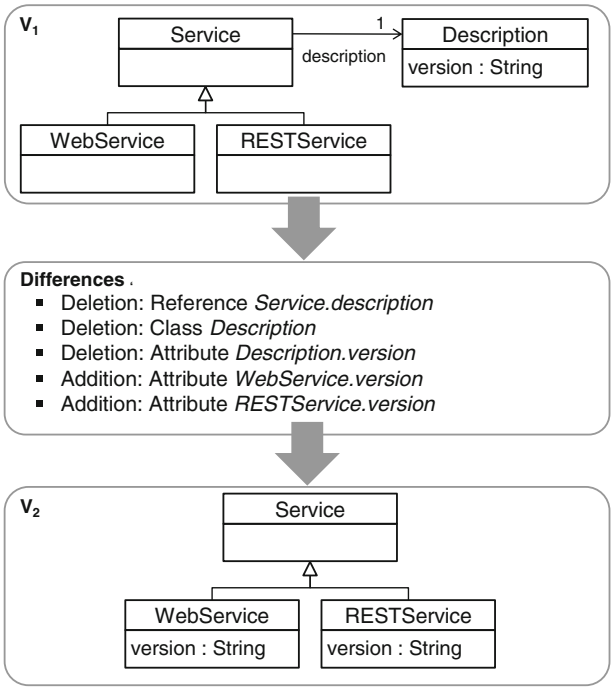

Fig. 1 Example for computing atomic differences

of atomic operations solely fails to reveal explicitly the reasons and intentions behind the actual evolution. Therefore, an explicit set of composite operations would be necessary.

Only a few approaches have been proposed that address also the detection of composite operations. These approaches search for patterns of atomic operations among all atomic operations obtained from model comparison tools. If a change pattern of a composite operation is found, they evaluate the pre- and post-conditions of the respective operation before an occurrence of the operation is reported. However, a detailed case study of such an approach (Langer et al. 2012) revealed that good results can be achieved as long as there are no overlapping sequences of composite operations; that is, the operations are applied in a sequence, whereas one operation is necessary before a subsequent operation can be applied. Moreover, in several scenarios the subsequent operations mask preceding operations (e.g., they delete an updated element so that the update is not visible anymore) or render the post-conditions of the preceding operations invalid. As a result, current approaches fail to detect overlapping sequences of composite operation sequences correctly. However, according to the case study presented in (Langer et al. 2012), around 20 % of all composite operations are applied within an overlapping operation sequence and hence are missed by current approaches.

2.2 Challenges of Detecting Composite Operations

To illustrate the issue of overlapping sequences of refactorings, we present in this section a motivating example based on the previously introduced example for illustrating atomic operations and discuss the challenges of finding an optimal sequence of composite operations that best describes the evolution a model has been subjected to.

2.2.1 Overlapping Sequences of Composite Operations

Consider the evolution scenario depicted in Fig. 2. In this scenario, we have two model versions, called *V1* and *V2*, in the repository. Between these two versions, two refactorings have been applied. First, the user applied the refactoring Inline Class. This refactoring moves all features of the class to be inlined into another class that originally references the inlined class. Finally, the inlined class can be deleted without losing any information. The precondition of this refactoring is that the inlined class is referenced through a single-valued reference by the class that eventually contains all features. Applying this refactoring to our example, the attribute version is moved from the class Description to Service and the class Description, as well as the reference description in class Service, is finally deleted.

In our example, the user further applied the refactoring Push Feature to the previously inlined attribute version. This refactoring adds a certain feature from a superclass to all its subclasses and finally removes the feature from the superclass. The precondition for this refactoring is that the class originally containing the feature is a superclass of the classes to which the feature is added. Thus, in our example, the attribute version is added to the subclasses of Service, namely WebService and RESTService. As all subclasses now contain this attribute, it can be removed from their superclass Service without affecting the external behavior of this model.

When applying a model comparison algorithm to the model versions V_1 and V_2, we detect that the class Description has been deleted and the attribute version has been added to WebService and RESTService. However, this situation does not match with any refactoring

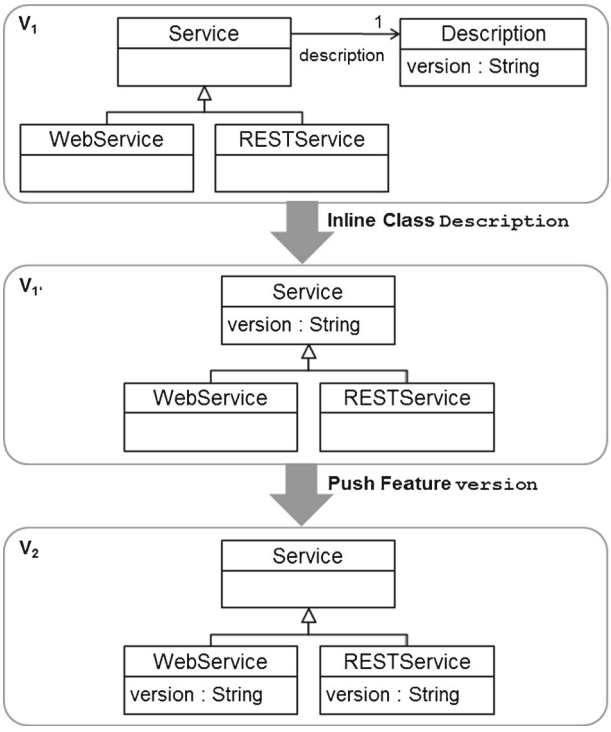

Fig. 2 A motivating example

specification, because neither the postconditions of Inline Class nor the preconditions of Push Feature are fulfilled when only considering the available model versions V_1 and V_2. Thus, no refactoring can be detected, as the second refactoring hides the operation constituting the first refactoring and the intermediate state V_1' is not available in the repository.

2.2.2 Quality of the Detected Operations Sequence

A naive approach to this problem would be to relax the refactorings' pre- and postconditions to a certain degree during the detection process. However, this would lead to several wrong refactoring indications (false-positives). Moreover, the pre- and postconditions are very specific to certain refactorings and it seems to be impossible to decide on a general basis which conditions can be relaxed and which conditions should hold at any time to accomplish a trustworthy refactoring detection.

A better approach is to search for a possible sequence of operations that lead from the initial model to the given final model, whereas the preconditions of every operation in this sequence are fulfilled when applied at the respective position in the sequence. However, as applying one operation in this sequence may affect the applicability of subsequent operations, finding a valid operation sequence is a combinatorial optimization problem within a huge search space. But not all valid solutions are good solutions in terms of understandability. Consider, for instance, the alternative operation sequence in Fig. 3, which constitutes a valid solution for the same initial version V_1 and revised version V_2 too. In this sequence, we first add the class Description as a superclass of Service and apply the composite operation Imitate Super Class to Service. This refactoring replaces a superclass by a subclass, whereas all its features are maintained by the respective subclass. Thus, in our example, the target of the reference description is changed from Description to Service and the attribute version of Description is moved to Service. Finally, the class Description is deleted as it is now *imitated* entirely by the class Service. Next, we delete the feature description and finally apply the Push Feature refactoring to version to obtain the same version V_2 as from the original sequence in Fig. 2.

Although this operation sequence is valid in terms of the preconditions of all applied operations and does not contain unnecessary operations (e.g., an addition of an element that is removed again later), it still does not reflect the evolution optimally, because it is not as concise as the original sequence and it still contains an atomic operation that has been originally part of a refactoring (i.e., Delete Feature, which deletes the reference description). Note that even more alternative sequences of operations that also constitute valid transitions from V_1 to V_2 are possible. However, all of these alternative operation sequences contain additional atomic and composite operations instead of showing the most direct transition from V_1 to V_2. In addition, it is easier for developers to understand changes described in terms of a short sequence of composite operations. Thus, these sequences distort the actual evolution, as well as the original intentions of the users. This makes it harder to understand the model evolution and potentially hampers the automation of certain model management tasks that depend on these operation sequences.

3 Model Evolution as a Multi-Objective Problem

This section shows how the above mentioned issues can be addressed using our multi-objective proposal and describes the principles that underlie the proposed method for detecting

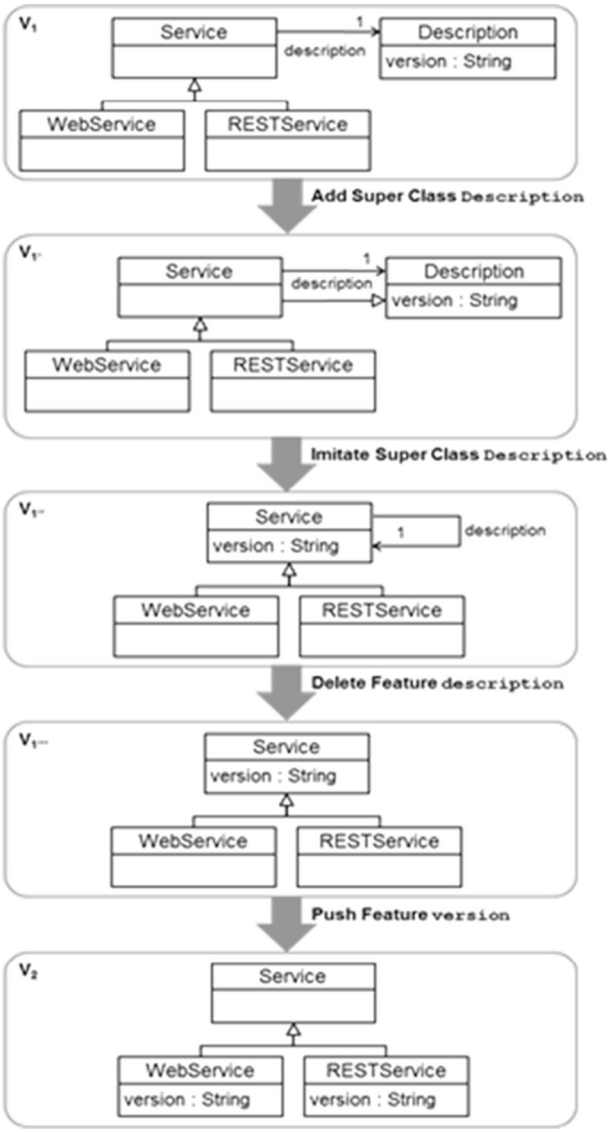

Fig. 3 Alternative operations sequence

refactorings from two versions of a model by maximizing detection correctness and minimizing the number of detected refactorings. Therefore, we first present an overview of the used NSGA-II algorithm and, subsequently, provide the details of the approach and our adaptation of the genetic algorithm to detect refactorings. The motivating example described in Section 2 is used to illustrate our approach described in the rest of this paper.

3.1 Approach Overview

The general structure of our approach is introduced in Fig. 4. The approach takes as input an initial and revised model, and as controlling parameters, an exhaustive list of refactoring

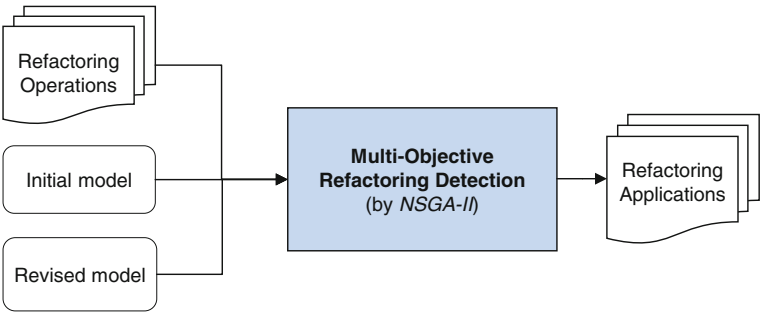

Fig. 4 Input and output of the multi-objective model change detection approach

operations. The approach generates a set of refactoring applications that represents the evolution from the initial model to the revised model. The process of detecting model changes can be viewed as the mechanism that finds the best way to combine refactoring operations of the exhaustive list of possible refactorings, in such a way to (*i*) maximize the similarity between the revised model and the resulting model when applying the detected refactorings on the initial model and (*ii*) minimize the number of refactorings. In other words, this process aims at finding the best tradeoff between these two conflicting criteria. In fact, maximizing the correctness of the detected refactorings corresponds to maximizing the similarity between the initial model after applying refactorings and the revised one (expected model). Minimizing the number of refactoring used to describe the changes corresponds to maximizing the use of a few larger refactorings instead of using a larger amount of smaller refactorings for describing the evolution.

Due to the large number of possible refactoring solutions and the two conflicting objectives, we consider the detection of refactoring between different model versions as a multi-objective optimization problem instead of a single-objective one. The algorithm explores a huge search space. In fact, the search space is determined not only by the number of possible refactoring combinations, but also by the order in which they are applied. Formally, if m is the number of available refactoring operations, then the number of possible refactoring solutions is given by all the permutations of all the possible subsets and is at least equal to: $(m!)^m$. This number can be huger, since the same refactoring operations can be applied several times in different model fragments. To explore this huge search space, we use the non-dominated sorting genetic algorithm (NSGA-II) to find the best trade-off between our two conflicting objectives (Deb et al. 2002). This algorithm and its adaptation to the refactoring problem are described in the next section.

3.2 Adapting Multi-Objective Optimization for Detecting Model Changes

This section describes how NSGA-II (Deb et al. 2002) can be used to detect high-level model changes by finding the suitable and minimal refactoring sequence. To apply NSGA-II to a specific problem, the following elements have to be defined: representation of the individuals, creation of a population of individuals, evaluation of individuals (using fitness functions) 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, and generation of a new population.

The next sections explain the adaptation of the design of these elements for our problem using NSGA-II.

3.2.1 NSGA-II Principles

Most real world optimization problems encountered in practice involve multiple criteria to be considered simultaneously (Deb et al. 2002). These criteria, also called objectives, are often conflicting. Usually, there is no single solution that is optimal with respect to all these objectives at the same time, but rather many different designs exist which are incomparable per se. Consequently, contrary to Single-objective Optimization Problems (SOPs) where we look for the solution presenting the best performance, the resolution of a multi-objective optimization (MOP) yields a set of compromise solutions presenting the optimal trade-offs between the different objectives. When plotted in the objective space, the set of compromise solutions is called the Pareto front.

The main goal in multi-objective optimization is to find a well-converged and well-distributed approximation of the Pareto front from which the decision maker (DM) will subsequently select his/her preferred alternative. A MOP consists in minimizing or maximizing an objective function vector under some constraints. The general form of a MOP is defined in (1) (Deb et al. 2002):

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

where f_i is the i^{th} objective function, 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_i satisfying the $(P + Q)$ constraints is said feasible and the set of all feasible solutions defines the feasible search space denoted by Ω. 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 yields a set of trade-off solutions, called Pareto optimal solutions or non-dominated solutions, and the image of this set in the objective space is called the Pareto front. Hence, the resolution of a MOP consists in approximating the whole Pareto front. In the following, we give some background definitions related to multi-objective optimization:

Definition 1.1 (*Pareto optimality*). A solution $x^* \in \Omega$ is Pareto optimal if and only if there does not exist another point $x \in \Omega$ such that $f_m(x) \leq f_m(x^*)$ for $\forall\ m \in I$ and $f_m(x) < f_m(x^*)$ for at least one $m \in I$.

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 1.2 (*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 1.3 (*Pareto optimal set*). For a given MOP $f(x)$, the Pareto optimal set is $P^* = \{x \in \Omega | \neg \exists\, x' \in \Omega,\ f(x') \preceq f(x)\}$.

Definition 1.4 (*Pareto optimal front*). For a given MOP $f(x)$ and its Pareto optimal set P^*, the Pareto front is $PF^* = \{f(x), x \in P^*\}$.

Figure 5 describes an example of traditional non-dominated solutions, ideal and nadir points.

In this paper, we adapted one of the widely used multi-objective algorithms called NSGA-II (Deb et al. 2002). NSGA-II is a powerful search method stimulated by natural selection that is inspired from the theory of Darwin (Koza 1992). Hence, the basic idea of NSGA-II 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 optimal solutions, called non-dominated solutions, also Pareto set. A non-dominated solution is the one which provides a suitable compromise between all objectives without degrading any of them. As described in Fig. 6, 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 an initial population R_0 of size N (line 5). As a consequence, NSGA-II starts by generating an initial population based on a specific representation that will be discussed later, using the exhaustive list of refactoring operations given as input as mentioned in the previous section. Thus, this population stands for a set of possible change detection solutions represented as sequences of refactorings which are randomly selected and combined.

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-dominated and will be selected by the NSGA-II to be one of the set of Pareto front. 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

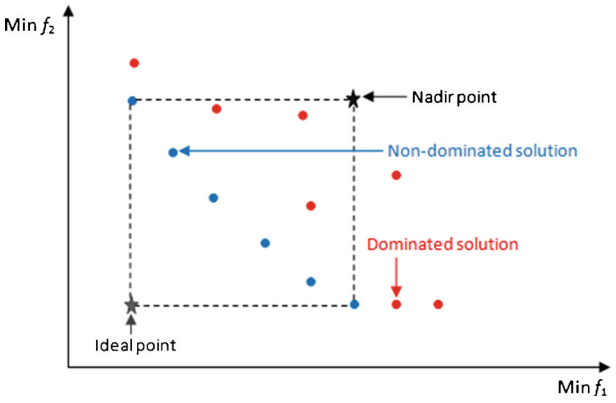

Fig. 5 An example of the nadir point and the ideal one for a bi-formulation

Fig. 6 NSGA-II overview

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** $|P_{t+1}| + |F_i| \leq N$ **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. $P_{t+1} = P_{t+1} \cup F_i[1 : (N\text{-}|P_{t+1}|)]$;
15. Q_{t+1} = create-new-pop(P_{t+1});
16. $t = t$+1;
17. **end**

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 (Deb et al. 2002) 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 stop criteria (line 4).

To summarize, the main NSGA-II loop goal is to make a population of candidate solutions evolve toward the best sequence of refactoring, i.e., the sequence that minimizes the number of refactoring and maximizes the similarity with the revised model version (expected model). During each iteration t, an offspring population Q_t is generated from a parent population P_t using genetic operators (selection, crossover and mutation). Then, Q_t and P_t are assembled in order to create a global population R_t. Then, each solution S_i in the population R_t is evaluated using our two fitness functions: (*i*) correctness function to maximize: consists to calculate the similarity score between the initial model after applying the refactoring solution and the revised one (expected model), and (*ii*) solution size function to minimize: represents the number of refactoring used to detect the changes.

The following three subsections describe more precisely our adaption of NSGA-II to the model change detection problem.

3.2.2 NSGA-II Adaptation

Solution Coding One key issue when applying a search-based technique is to find a suitable mapping between the problem to solve and the techniques to use, i.e., in our case, detecting high-level model changes. Figure 7 shows an example where the ith individual (solution) represents a

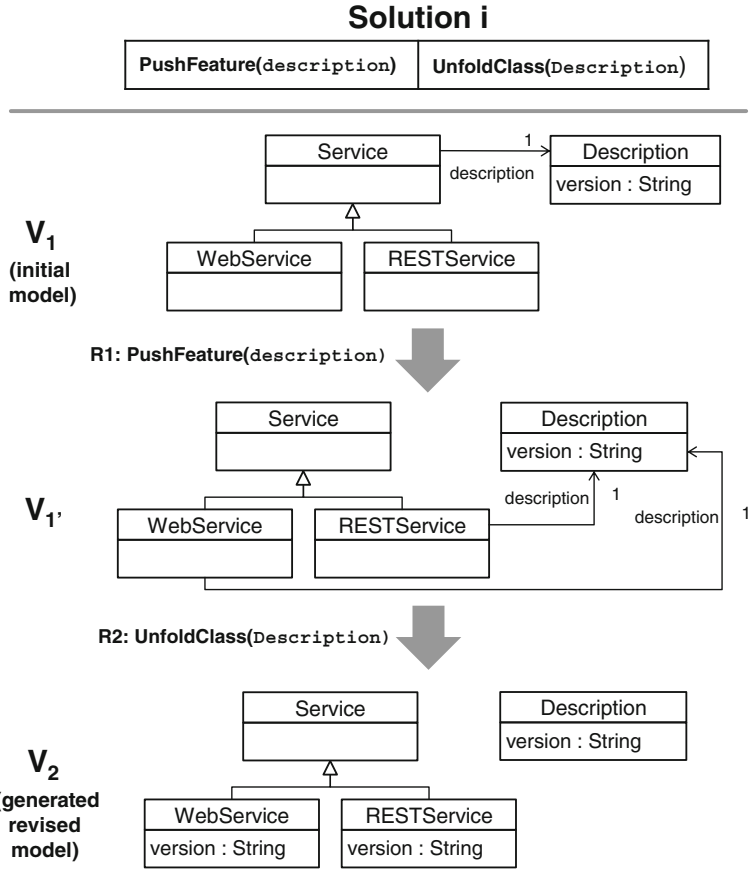

Fig. 7 Solution coding example

combination of refactoring operations to apply. The order of applying refactorings corresponds to their position in the vector (referred to as dimension number in the following). In addition, the execution of the refactorings is respecting pre- and postconditions to avoid conflicts and semantic inconsistencies. Furthermore, it has to be noted that the same type of refactoring operation could be applied several times in the same solution (but to different model fragments).

For instance, the solution represented in Fig. 7 is composed by two dimensions corresponding to two refactoring operations to apply in some model fragments of Fig. 1. The two refactorings composing the generated solution are *Push* Feature that is applied to the reference *description* and *Unfold Class* that is applied to the class *Description*. As described in Fig. 7, after applying the proposed solution we obtain a new model that will be compared with the expected revised model using a fitness function.

When generating a solution, we selected from the exhaustive list of refactorings proposed by Fowler (Fowler et al. 1999) only those that can be applied at the model level for UML class diagrams. But it has to be noted that the approach is designed to be generic, thus also other refactorings for various modeling languages may be supported.

Initial Population Generation To generate an initial population, we start by defining the maximum vector length including the number of refactorings. The vector length is

proportional with the number of refactorings to use for detecting model changes. Sometimes, a high vector length does not mean that the results are more precise, but that only a few refactorings are sufficient to detect changes. These parameters can be specified either by the user or chosen randomly. Thus, the individuals have different vector length (structure). Then, for each individual we randomly assign one refactoring, with its parameters, to each dimension.

Since any refactoring combination is possible, we do need to define some conditions to verify when generating an individual. For example, we cannot remove a method from a class that was already moved to another class, thus the pre-conditions and postconditions of the operations have to be fulfilled before their execution.

Genetic Operators We are using the following operators.

Selection To select the individuals that will undergo the crossover and mutation operators, we used the stochastic universal sampling (SUS) (Fonseca and Fleming 1993), in which the probability of selection of an individual is directly proportional to its relative fitness in the population. SUS is a random selection algorithm which gives higher probability to be selected to the fittest solutions while still giving a chance to every solution. For each iteration, we use SUS to select individuals (population_size/2) from population P_n for the next population P_{n+1}. These selected individuals (upper half of the ranking) will "give birth" to new individuals (substituting the lower half of the ranking) using the crossover operator.

Crossover When two parent individuals are selected, a random cut point is determined to split them into two sub-vectors. The crossover operator selects a random cut-point in the interval [0, min_{length}] where min_{length} is the minimum length between the two parent chromosomes. Then, crossover swaps the sub-vectors from one parent to the other. Thus, each child combines information from both parents. This operator must enforce the length limit constraint by eliminating randomly some refactoring operations.

For each crossover, two individuals are selected by applying the SUS selection. Even though individuals are selected, the crossover happens only with a certain probability.

The crossover operator allows creating two offspring P_1' and P_2' from the two selected parents P_1 and P_2. It is defined as follows. A random position k is selected. The first k refactorings of P_1 become the first k elements of P_1'. Similarly, the first k refactorings of P_2 become the first k refactorings of P_2'. Figure 8 shows an example of the crossover process. In this example, P_1 and P_2 are combined to generate two new solutions. The right sub-vector of

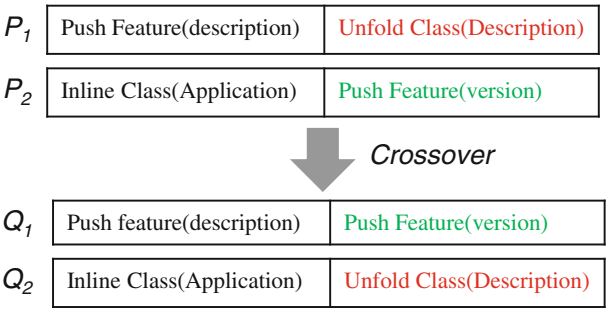

Fig. 8 A crossover operator applied to an example of refactoring solutions

P_1 is combined with the left sub-vector of P_2 to form the first child, and the right sub-vector of P_2 is combined with the left sub-vector of P_1 to form the second child. In Fig. 8, the position k takes the value 1. The second refactoring of P_1 becomes the second element of P_2. Similarly, the second refactoring of P2 becomes the second refactoring of P_1.

Mutation The mutation operator consists of randomly changing one or more dimensions (refactoring) in the solution (vector). Given a selected individual, the mutation operator first randomly selects some positions in the vector representation of the individual. Then the selected dimensions are replaced by other refactoring. Furthermore, the mutation can only modify the controlling elements of some positions without replacing the operation by a new one. Figure 9 illustrates the effect of a mutation that replaced the dimension number one *Push Feature* (*description*) by *Push Feature* (*version*).

When creating a sequence of refactorings (an individual), it is important to guarantee that they are feasible and that they can be legally applied. The first work in the literature was proposed by (Opdyke 1992) who introduced a way of formalizing the preconditions that must be met before a refactoring can be applied and ensure that the behavior of the system is preserved. Opdyke created functions which could be used to formalize constraints. These constraints are similar to the Analysis Functions used later by (O Cinnéide M 2001) and (Roberts and Johnson 1999). The reader may find more details about the different constraints in these references (Opdyke 1992; O Cinnéide M 2001; Roberts and Johnson 1999). In addition, we define in the following Table 1 these pre- and post-conditions for most of the refactorings used by our approach (Section 3.2.2):

When applying the mutation and crossover, we used also a repair operator to delete duplicated refactorings after applying the crossover and mutation operators.

Multi-Criteria Evaluation (a.k.a Fitness Functions) In our previous work (Ben Fadhel et al. 2012), we used one objective (fitness function) for realizing search-based detection of model changes. In this work, we are using two different fitness functions in our NSGA-II adaptation.

In general, the encoding of an individual should be formalized as a mathematical function called "fitness function". The fitness function quantifies the quality of the proposed refactoring sequence. The goal is to define an efficient and simple fitness function in order to reduce the computational complexity. In our work, we are using two fitness functions. The first function is based on a similarity score between the generated models and expected ones to maximize the correctness of detected model changes. The second function, to minimize the size of the refactoring sequence, counts the number of used refactoring to detect the changes. In fact, minimizing the number of refactoring improves the comprehension of model evolution by favoring course-grained changes over fine-grained changes and reduces possible redundancies in the change sequence.

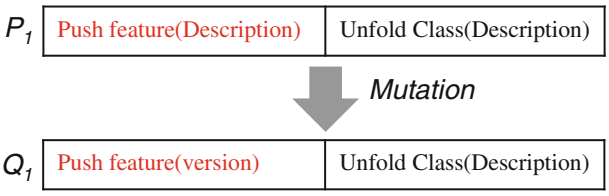

Fig. 9 A mutation operator applied to an example of refactoring solutions

Table 1 Refactorings pre- and post-conditions

Refactorings	Pre and post-conditions
Move method(c1,c2,m)	Pre: exist(c1, c2, m) AND isClass(c1, c2) AND isMethod(m) AND NOT(inheritanceHierarchy(c1,c2)) AND defines(c1,m) AND NOT(defines (c2, sig(m)) Post: exist(c1, c2, m) AND defines(c2,m) AND NOT(defines(c1, m))
Move field(c1,c2,f)	Pre: exist(c1, c2, f) AND areClasses(c1, c2) AND isField(f) AND NOT (inheritanceHierarchy(c1,c2)) AND defines(c1,f) AND NOT (defines(c2, f)) Post: exist(c1, c2, f) AND defines(c2,f) AND NOT(defines(c1, f))
Pull up field(c1,c2,f)	Pre: exist(c1, c2, f) AND areClasses(c1, c2) AND isField(f) AND isSuperClassOf(c2,c1) AND defines(c1,f) AND NOT(defines(c2, f)) Post: exist(c1, c2, f) AND defines(c2,f) AND NOT(defines(c1, f))
Pull up method(c1,c2,m)	Pre: exist(c1, c2, m) AND areClasses(c1, c2) AND isMethod(m) AND isSuperClassOf(c2,c1) AND defines(c1,m) AND NOT(defines (c2, sig(m)) Post: exist(c1, c2, m) AND defines(c2,m) AND NOT(defines(c1, m))
Push down field(c1,c2,f)	Pre: exist(c1, c2, f) AND areClasses (c1, c2) AND isField(f) AND isSubClassOf(c2,c1) AND defines(c1,f) AND NOT(defines(c2, f)) Post: exist(c1) AND exists(c2) AND exits(m) AND defines(c2,m) AND NOT(defines(c1, m))
Push down method(c1,c2,m)	Pre: exist(c1, c2, m) AND areClasses (c1,c2) AND isMethod(m) AND isSubClassOf(c2,c1) AND defines(c1,m) AND NOT(defines(c2, sig(m)) Post: exist(c1, c2, m) AND defines(c2,m) AND NOT(defines(c1, m))
Inline class(c1,c2)	Pre: exist(c1, c2) AND areClasses (c1, c2) Post: exists(c1) AND NOT(exists(c2))
Extract class(c1,c2)	Pre: exists(c1) AND NOT(exists(c2)) AND isClass(c1) AND $\|methods(c1)\| \geq 2$ Post: exist(c1, c2) AND isClass(c2)
Extract super class(c1,c2)	Pre: exists(c1) AND NOT(exists(c2)) AND isClass(c1) AND $\|methods(c1)\| \geq 2$ Post: exists(c1) AND exists(c2) AND isClass (c2) AND isSuperClass(c1,c2)
Extract sub class(c1,c2)	Pre: exists(c1) AND NOT(exists(c2)) AND isClass(c1) AND $\|methods(c1)\| \geq 2$ Post: exist(c1, c2) AND isClass(c2) AND isSubClass(c1,c2)

Where,

— isClass(c): c is a class (similarly for areClasses())

— isInterface(c): c is an interface (similarly for areInterfaces())

— isMethod(m): m is a method

— Sig(m): the signature of the method m

— isField(f): f is a field

— defines(c,e): the code element e (method or field) is implemented in the class/interface c

— exists(e): the code element e exists in the current version of the code model (Similarly for exist())

— inheritanceHierarchy(c1,c2): both classes c1 and c2 belong to the same inheritance hierarchy

— isSuperClassOf(c1,c2): c1 is a superclass of c2

— isSubClassOf(c1,c2): c1 is a subclass of c2

— fields(c): returns the list of fields defined in the class or interface c

— methods(c): returns the list of methods implemented in class or interface c

a) Correctness function.

The quality of an individual, in terms of correctness, is proportional to the quality of the refactoring operations composing it. In fact, the execution of these refactorings, modifies

various model fragments. Then, the quality of a solution is determined with respect to the expected revised model. In other words, the best solution is the one that maximizes the similarity between the computed model, obtained after applying the refactoring sequence, and the expected revised model.

In this context, we define the fitness function of a solution, normalized in the range [0, 1], as:

$$f_{norm} = \frac{\frac{\sum_{i=1}^{p} a_i}{t} + \frac{\sum_{i=1}^{p} a_i}{p}}{2} \quad (1)$$

where t is the number of model elements in the expected revised model, p is the number of model elements in the generated model, and a_i has value 1 if the i^{th} element in the generated model exists in the expected one, and value 0 otherwise. The definition of a model element is not limited to the classes or methods but also includes all the possible relationships between classes.To illustrate the fitness function, we consider the example of Figs. 2 and 7. The model V_2 of Fig. 3 is considered as the expected revised model and V_2 of Fig. 7 as the generated model after applying the refactoring solution (Fig. 10). There are two differences between the two models which correspond to class *Description* and attribute *version*. Thus, the fitness function in this case is:

$$f_i = \frac{\frac{6}{6} + \frac{6}{8}}{2} = 0.875 \quad (2)$$

The outcome of the search is a set of acceptable solutions that may generate approximately the expected model when applied to the initial one with a minimum number of refactorings. Thus, it happens, of course, that the best solution does not generate the exact expected model. In our formulation, we considered that it is maybe important to penalize the solutions that did not generate the expected model and thus missed some operations (that could be just adding a new method or a new attribute or deleting an element) to highlight to the software designer that not only refactorings were applied between the two releases. This information can be used by the software designer to identify these missing atomic changes. In addition, it can be an

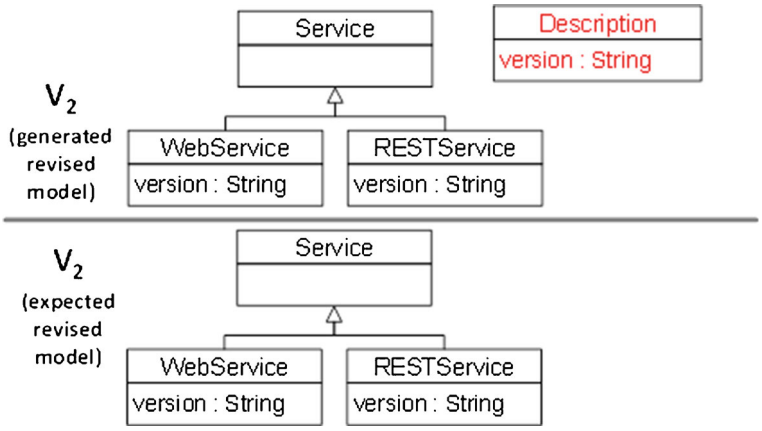

Fig. 10 Comparison between the generated model and the expected model (cf. Fig. 3)

interesting future research direction to combine the detection of both atomic changes and refactorings in our approach by maximizing the number of refactorings and minimizing the number of atomic changes. In this case, most of the combinations of atomic changes that may lead to refactorings will be performed to maximize the first objective and the remaining atomic changes will correspond to added or deleted elements but not refactorings.

b) Solution-size function.

In our approach, we define the complexity function, to minimize, as the number of refactoring used to detect the changes:

$$f_{size} = n \tag{3}$$

where n is the number of refactorings in the generated solution. For instance, the size of the solution generated in Fig. 7 is 2.

4 Evaluation

In order to evaluate the feasibility of our approach for detecting high-level model changes, we conducted an experiment based on different versions of real-world models extracted from large open source systems and one industrial project.

4.1 Research Questions

We defined seven research questions that address the applicability, performance in comparison to existing model changes detection approaches, and the usefulness of our multi-objective formulation. The seven research questions are as follows:

- RQ1: Search validation (sanity check). To validate the problem formulation of our approach, we compared our NSGA-II formulation with the simple deterministic greedy algorithm (GR) (Ruiz and Stützle 2007 Mar 16). GR starts with an empty set of refactoring operations and at each generation it adds the refactoring operation providing the maximum improvement in terms of similarity (first objective); the search stop when any further operation will not improve the similarity objective. Clearly, the refactoring operations to try at each iteration are only those operations related to the part of the model that have been changed between two subsequent releases. This greedy algorithm is very simple and often very efficient when dealing with "minimization" problem where one of the goal is to minimize the number of elements selected from a set (the number of refactoring operations selected from the set of potential operations that could be performed on the changed part of the model). If GR outperforms an intelligent search method, we can conclude that there is no need to use a metaheuristic search.
- RQ2: To what extent can the proposed approach detect correctly changes between different model versions (in terms of correctness and completeness) and reduce the number of refactorings (reducing redundancies/overlaps and maximizing the use of complex/composite refactorings)?

The next four questions are related to the comparison between our proposal and the state-of-the-art model changes detection approaches.

- RQ3.1: How does NSGA-II perform compared to another multi-objective algorithm? It is important to justify the use of NSGA-II for the problem of model changes detection. We compare NSGA-II with another widely used multi-objective algorithm, Multi-Objective Particle Swarm Optimization (MOPSO), (Li 2003) using the same fitness functions.
- RQ3.2: How does our multi-objective model changes detection formulation perform compared to a mono-objective one? A multi-objective algorithm provides a trade-off between the two objectives where developers can select their desired changes detection solution from the Pareto-optimal front. A mono-objective approach uses a single fitness function that is formed as an aggregation of both objectives and generates as output only one solution of detected changes. We converted the second objective (number of refactoring) from a minimization problem to a maximization one using the traditional equation: $F(x) = 1/f\_\{size\}$, where $f\_\{size\}$ is the number of refactoring operations (see Equation 3). Thus, the two objectives are to maximize and normalized in the range of [0..1]. The aggregated fitness function is to maximize and represent the average of the two sub-functions. This comparison is required to ensure that the solutions provided by NSGA-II and MOPSO provide a better trade-off between the two objectives than a mono-objective approach. Otherwise, there is no benefit to our multi-objective adaptation.
- RQ3.3: How does NSGA-II perform compared to existing search-based model changes detection approaches? Our proposal is the first work that treats the problem of model changes detection using a multi-objective approach. However, in our previous work (Ben Fadhel et al. 2012), we used a mono-objective genetic algorithm for model changes detection using only one objective which is maximizing the similarity with the expected version.
- RQ3.4: How does NSGA-II perform compared to existing model changes detection approaches not based on the use of metaheuristic search? While it is very interesting to show that our proposal outperforms existing search-based model changes detection approaches, developers will consider our approach useful, if it can outperform other existing tools (Langer et al. 2012; Xing and Stroulia 2005) that are not based on optimization techniques.

The last research question is related to the benefits of our approach for software engineers.

- RQ4 (Insight): Can our multi-objective approach for model changes detection be useful for software engineers? A feed-back from software engineers is required to illustrate the importance of the use of a multi-objective approach for model changes detection in a real-world setting. We asked all subjects of our experiments, including the developers from our industrial partner, to take a study questionnaire to collect their opinions about our model changes detection tool and its relevance.

4.2 Experimental Setting

We analyzed the revisions of three models, namely the Graphical Definition Metamodel, the Generator Metamodel, and the Mappings Metamodel. Therefore, the respective metamodel

versions had to be extracted from GMF's version control system and, subsequently, manually analyzed to determine the actual applied changes between successive metamodel versions. The manual analysis of the extracted changes and the conversion into refactorings involved a total number of 18 subjects divided into 3 groups (6 subjects each) (http://www-personal.umd.umich.edu/~marouane/modelchanges/ and https://figshare.com/articles/Model_Changes_Data/3180478). All the subjects are volunteers and familiar with Java development and Model Driven engineering (MDE). The experience of these subjects on MDE and Java programming ranged from 2 to 15 years. The participants who evaluated the open source models have a good knowledge about these systems and they did similar experiments in the past on the same systems. We also selected the groups based on their familiarity with the studied models. The groups included four undergraduate students, six master students, seven PhD students and one faculty member. Three of the master students are also working at General Motors as senior software engineers. Subjects were familiar with the practice of refactoring, and model changes detection. A peer-validation was performed between the three groups on the three models to check the validity of the manually defined refactorings between the three versions. In addition to these 18 subjects, we asked a set of 13 software developers from the IT department of our industrial partner the Ford Motor Company to participate in a study questionnaire to evaluate the relevance of our tool as detailed later. We selected these 13 developers because they are familiar with our work related to software refactoring based on different funded projects. Furthermore, 7 out the 13 developers from Ford participated in the validation of the results on one of their projects (MROI).

For achieving a broader data basis, we additionally used an existing corpus (Langer et al. 2012; Ben Fadhel et al. 2012) of 81 releases of four open source Java projects, namely Apache Ant, Gantt, JHotdraw and Xerces-J. Apache Ant is a build tool and library specifically conceived for Java applications. In (Cicchetti and Di Ruscio 2007; Ben Fadhel et al. 2012), the authors extracted and validated the different refactorings between the different versions by asking groups of students and programmers. Gantt Project includes several project management functions like designing a Gantt chart for the scheduling of tasks. Xerces is a family of software packages for parsing and manipulating XML. It implements a number of standard APIs for XML parsing. JHotdraw is a framework used to build graphic editors. We considered the evolution across three versions of Gantt (v1.7, v1.8, and v1.9.10), three versions of JHotDraw (v5.1, v5.2, and v5.3), four versions of ApacheAnt (v1.6.1, v1.6.2, v1.6.3 and v1.6.4) and four versions of Xerxes-J (v1.4.4, v2.5.0, v2.6.0, and v2.6.1). The number of refactorings considered in our experiments is 659 that can be considered on model level and, thus, constitute the input of our experiment for the open source systems.

We choose Xerces-J, Gantt, JHotDraw, and Apache Ant because they are medium-sized open-source projects and were analyzed in the related work. The initial version of Gantt, GMF, and Apache Ant was known to be of poor quality, which has led to a new major revised version. Xerces-J, Gantt, JHotDraw, and Apache Ant have been actively developed over the past 10 years, and their design has not been responsible for a slowdown of their developments.

In order to get feedback from the original developers of a system, we considered in our experiments a large industrial project, called MROI, provided by our industrial partner the Ford Motor Company. MROI is a marketing return on investment tool used by the marketing department of Ford to estimate the sales of cars based on the demand, dealers' information, advertisements, etc. The tool can collect, analyze and synthesize a variety of data types and sources related customers and dealers. It was implemented during a period of more than 8 years and frequently changed to include and remove several features. In total, 327 refactoring

has been applied between the different releases since we selected only those that can be applied to the model level when using UML class diagram based modeling languages. The evolution of the different models provides a relatively large set of revisions. In total, the evolution of the considered models comprises 63 revisions that involved at least one refactoring operation. Most of the commits comprise between 1 and 14 refactoring operations.

We did not restrict the participants to only use the models when building the oracle, but they can use the source code in case of uncertainty or ambiguities when identifying and evaluating the changes. Table 2 describes the number of expected refactorings to be detected by our approach on the different models.

When comparing two mono-objective algorithms, it is usual to compare their best solutions found so far during the optimization process. However, this is not applicable when comparing two multi-objective evolutionary algorithms since each of them gives as output a set of non-dominated (Pareto equivalent) solutions. For this reason, we use the three following performance indicators (Deb et al. 2002) when comparing NSGA-II and MOPSO:

- *Hypervolume* (*IHV*) corresponds to the proportion of the objective space that is dominated by the Pareto front approximation returned by the algorithm and delimited by a reference point. Larger values for this metric mean better performance. The most interesting features of this indicator are its Pareto dominance compliance and its ability to capture both convergence and diversity. The reference point used in this study corresponds to the nadir point (Bechikh et al. 2010) of the Reference Front (*RF*), where the Reference Front is the set of all non-dominated solutions found so far by all algorithms under comparison.
- *Inverse Generational Distance* (*IGD*) is a convergence measure that corresponds to the average Euclidean distance between the Pareto front Approximation (*PA)* provided by the algorithm and the reference front *RF*. The distance between *PA* and *RF* in an *M*-objective space is calculated as the average *M*-dimensional Euclidean distance between each solution in *PA* and its nearest neighbour in *RF*. Lower values for this indicator mean better performance (convergence).
- *Contribution* (*IC*) corresponds to the ratio of the number of non-dominated solutions the algorithm provides to the cardinality of *RF*. Larger values for this metric mean better performance.

We note that the mono-objective algorithm provides only one solution of detected changes, while NSGA-II and MOPSO generate a set of non-dominated solutions. In order to make

Table 2 The systems studied

Model	Number of expected refactoring	Number of model elements (min, max)
GMF Graph	36	277,310
GMF Gen	112	883,1295
GMF Map	14	367, 428
GanttProject	72	451, 572
Xerces-J	86	1698,1732
JHotDraw	81	985, 1457
ApacheAnt	258	2166, 2489
MROI-Ford	327	2481, 2704

meaningful comparisons, we select the best solution for NSGA-II and MOPSO using a knee point strategy (Zitzler et al. 2003). The knee point corresponds to the solution with the maximal trade-off between minimizing the number of refactorings and maximizing the similarity with the expected model version. Hence moving from the knee point in either direction is usually not interesting for the user. Thus, for NSGA-II and MOPSO, we select the knee point using the trade-off "worth" metric proposed by Rachmawati & Srinivasan (Zitzler et al. 2003) that corresponds to the solution having the median IHV value. We aim by this strategy to ensure fairness when making comparisons against the mono-objective approaches. For the latter, we use the best solution corresponding to the median observation on 51 runs. We used 51 runs since it is easy to select the median run. This metric estimates the worthiness of each non-dominated changes detection solution in terms of trade-off between our two conflicting objectives. After that, the knee point corresponds to the solution having the maximal trade-off "worthiness" value.

The quality of our results was measured by two methods: automatic correctness (AC) and manual correctness (MC). Automatic correctness consist of comparing the detected changes to the reference ones, operation by operation using precision (AC-P) and recall (AC-R). AC method has the advantage of being automatic and objective. However, since different refactoring combinations exist that describe the same evolution (different changes but same target model), AC could reject a good solution because it yields different refactoring operations from reference ones. To account for those situations, we also use MC which manually evaluates the detected changes, here again operation by operation. We calculate also NR the number of refactoring used to detect the changes between the different model versions.

We compared our results with an existing work proposed by Langer et al. (Langer et al. 2012) that introduced a deterministic approach for defining and detecting composite operations for software models based on model transformations by-example techniques. The approach allows defining composite operations on models by demonstrating them on examples from which the general transformations (including the pre- and post-conditions of the operations expressed as OCL constraints) is semi-automatically derived. These transformations can be used for applying the composite operations as well as detecting applications of them between two versions of a model. A composite operation is detected either if it's pre- and postconditions are fulfilled in the initial or if the pre-conditions can be established by an already detected composite operation, but the postconditions always have to be fulfilled by the revised version for all detected composite operations. For more information on the approach and its evaluation, we refer the interested reader to (Langer et al. 2012).

We also compared our approach to UMLDiff a deterministic refactoring detection technique not based on heuristic search (Xing and Stroulia 2005) using the default parameters of the tool. UMLDiff includes three main steps. First, facts regarding design-level entities and their relations in each individual version of the system are extracted from the source-code versions of the system. Then, the designs of subsequent system versions are pair-wise compared to determine how the basic entities and relations have changed from one version to the next. Finally, queries, corresponding to typical refactorings' change patterns, are applied to the change-facts database to extract instances of particular refactorings and their participant elements.

Our experiments include also a comparison of our multi-objective approach to our previous work where only a mono-objective genetic algorithm is used to maximize changes correctness (Ben Fadhel et al. 2012). Furthermore, we compared the performance of the simple deterministic greedy algorithm and another multi-objective algorithm, multi-objective genetic algorithm, to NSGA-II. We used the AC scores for all these comparisons.

4.3 Statistical Tests

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 Friedman test (Arcuri and Briand 2011; Friedman 1937) with a 95 % confidence level ($\alpha = 5$ %). The Friedman test is a non-parametric statistical test useful for multiple pairwise comparisons. The latter verifies the null hypothesis H_0 that the obtained results of the different algorithms are samples from continuous distributions with equal medians, as against the alternative that they are not, H_1. The p-value of the Friedman 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 (≤ 0.05) means that we accept H_1 and we reject H_0. However, a p-value that is strictly greater than α (>0.05) means the opposite. In fact, we compute the p-value of GR, MOPSO and mono-objective search results with NSGA-II ones. In this way, we could decide whether the superior performance of NSGA-II to one of each of the others (or the opposite) is statistically significant or just a random result.

The Friedman test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference in magnitude. To this end, we used the Vargha and Delaney's A statistics (Arcuri and Briand 2011) which is a non-parametric effect size measure. In our context, given the different performance metrics (such as Precision, Recall and NR), the A statistics measures the probability that running an algorithm B1 (NSGA-II) yields better performance than running another algorithm B2 (such as MOPSO, Agg-GA and GA). If the two algorithms are equivalent, then $A = 0.5$.

4.4 Parameter Setting and Tuning

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 (Table 3), we perform a set of experiments using several population sizes: 100, 150, 200, 400 and 500. The maximum length of the chromosome is limited to 600. The stopping criterion was set to 100,000 evaluations for all algorithms in order to ensure fairness of comparison. The MOPSO used in this paper is the Non-dominated Sorting PSO (NSPSO) proposed by Li (Li 2003). The other parameters' values were fixed by trial and error (Eiben and Smit 2011) and are as follows: (1) crossover probability = 0.6; mutation probability = 0.4 where the probability of

Table 3 Best population size configurations

System	NSGA-II	MOPSO	Agg-GA	GA	GR
GMF Map	150	200	200	200	200
GMF Graph	150	200	300	200	200
GMF Gen	200	200	300	300	300
GanttProject	200	300	400	400	300
Xerces-J	200	300	400	400	400
JHotDraw	400	500	500	500	500
ApacheAnt	400	500	500	500	500
MROI-Ford	400	500	500	500	500

gene modification is 0.2; stopping criterion = 100,000 evaluations. For MOPSO, the cognitive and social scaling parameters c_1 and c_2 were both set to 1.5 and the inertia weighting coefficient w decreased gradually from 1.0 to 0.3. The maximum length of the individuals is limited to the size of the evaluated model (number of model elements). In fact, we found, based on the used trials and error method, that the number of the detected changes does not exceed, in general, the total number of model elements (equivalent to the number of changes to create a model from scratch).

4.5 Results and Discussions

4.5.1 Results for RQ1

To answer the first research question RQ1, we implemented the simple deterministic greedy algorithm. The pre- and post-conditions discussed in Section 3 were used to check the feasibility of generated refactoring solutions. The obtained Pareto fronts were compared for statistically significant differences with NSGA-II using *IHV*, *IGD* and *IC*.

Table 4 confirms that both multi-objective algorithms NSGA-II and MOPSO are better than GR search based on the three quality indicators IHV, IGD and IC on all the seven open source systems and the industrial project. The Friedman test showed that in 51 runs both NSGA-II and MOPSO results were significantly better than GR. Table 4 shows the overview of the results of the significance tests comparison between NSGA-II and MOPSO. NSGA-II outperforms MOPSO in most of the cases: 17 out of 24 experiments (71 %). MOPSO outperforms the NSGA-II approach only in GMF Gen and GMF Graph, which are the smallest open source system considered in our experiments, having a low number of operations to detect. In particular, NSGA-II outperforms MOPSO in terms of IHV values in 7 out 8 experiments with one 'no significant difference' result. However, the results of NSGA-II for Apache Ant are very comparable to MOPSO in terms of IGD and IC. In fact, one the main reasons is that Apache Ant has the highest number of expected changes to be detected (258 refactorings) which is almost more than the double of refatorings to detect in the remaining models. The high number of changes introduced to the Apache Ant releases makes the search process much more challenging and complicated due to the very large number of possible refactoring combinations. MOPSO performance is very similar and sometimes better than NSGA-II when exploring the largest search space.

Table 4 The significantly best algorithm among GR, NSGA-II and MOPSO (Not sign. diff. means that NSGA-II and MOPSO are statistically equivalent, but both the two algorithms are statistically better than GR search)

System	*IHV*	*IGD*	*IC*
GMF Map	NSGA-II	NSGA-II	NSGA-II
GMF Graph	NSGA-II	*MOPSO*	*MOPSO*
GMF Gen	NSGA-II	*MOPSO*	*MOPSO*
GanttProject	NSGA-II	NSGA-II	NSGA-II
Xerces-J	NSGA-II	NSGA-II	NSGA-II
JHotDraw	*Not sign. diff.*	NSGA-II	NSGA-II
ApacheAnt	NSGA-II	*Not sign. diff.*	*Not sign. diff.*
MROI-Ford	NSGA-II	NSGA-II	NSGA-II

Table 4, Figs. 12 and 13 confirm the superior performance of NSGA-II and MOPSO comparing to GR in terms of automatic and manual correctness and number of refactorings. All these results were statistically significant on 51 independent runs using the Friedman test with a 95 % confidence level ($\alpha < 5$ %). We have also found that NSGA-II is better than GR algorithms based on all the performance metrics with an A effect size higher than 0.91 on all the systems.

GR is not efficient to generate good changes detection solutions using all the above metrics in all the experiments. Thus, an intelligent algorithm is required to find good trade-offs to propose efficient solutions. We conclude that there is empirical evidence that our multi-objective formulation surpasses the performance of GR search thus our formulation is adequate and the use of metaheursitic search is justified (this answers RQ1).

4.5.2 Results for RQ2

To answer RQ2, we evaluated the average of NR, AC-P, AC-R and MC scores for non-dominated changes detection solutions proposed by NSGA-II.

Table 5 confirms that our NSGA-II adaptation was successful in detecting model changes and describing them with a minimum number of refactorings. Overall, our approach was able to detect 932 refactoring operations correctly among all the 1037 expected operations (i.e., around 90 % of recall), whereas 986 refactorings have been detected, which leads to a precision of around 94 %. It is worth noting that the evolution history of the different systems is very different. The Graphical Definition Metamodel (GMF Graph for short) was extensively modified within only one large revision comprising 36 refactoring operations, but most of them were detected using our technique with a good recall of 80 %. The Generator Metamodel (GMF Gen for short) was subjected to 40 revisions. Thus, the evolution of this model is a very representative mixture of different scenarios for the detection of refactoring operations leading

Table 5 NSGA-II detection correctness

Model	AC-Precision	AC-Recall	MC	NR
GMF Map	14/14 = 100 % STDev = 1.52	14/17 = 82 % STDev = 1.18	14/17 = 82 % STDev = 0.89	17 STDev = 1.08
GMF Graph	31/36 = 77 % STDev = 1.24	31/39 = 79 % STDev = 0.82	33/39 = 91 % STDev = 1.14	39 STDev = 1.43
GMF Gen	104/112 = 92 % STDev = 1.82	104/122 = 85 % STDev = 0.94	108/122 = 88 % STDev = 1.12	122 STDev = 2.58
GanttProject	69/72 = 96 % STDev = 1.34	69/84 = 82 % STDev = 0.79	69/84 = 87 % STDev = 1.02	84 STDev = 2.72
Xerces-J	81/86 = 96 % STDev = 1.16	81/92 = 88 % STDev = 1.12	86/94 = 91 % STDev = 1.21	92 STDev = 1.89
JHot-draw	77/81 = 95 % STDev = 1.38	77/86 = 89 % STDev = 0.91	82/93 = 88 % STDev = 1.78	86 STDev = 2.87
Apache Ant	242/258 = 92 % STDev = 1.59	242/267 = 90 % STDev = 1.19	254/267 = 95 % STDev = 1.01	267 STDev = 2.94
MROI-Ford	314/327 = 96 % STDev = 2.11	314/331 = 95 % STDev = 1.82	328/341 = 96 % STDev = 1.72	330 STDev = 2.42

Median number of disabled refactorings on 51 independent runs. The results were statistically significant on 51 independent runs using the Friedman test with a 95 % confidence level ($\alpha < 5$ %)

to a precision of 92 % and a recall of 85 %. The evolution of the third model under consideration, the Mappings Metamodel (GMF Map for short), contained four revisions and in each revision at maximum five refactoring operations have been applied that all of them were detected using our approach.

As the studied evolution of the remaining systems covers a time period of ten years (as opposed to the considered time period of two years of the GMF case study), they have a larger number of refactorings to be detected. However, our multi-objective proposal performs also well on the large evolutions of the remaining systems. For Apache, most of detected changes are correct with 92 % and 90 % respectively as precision and recall. For GanttProject and JHotDraw, they have approximately similar number of refactoring to detect and our approach detected most of them with an average of more than 92 % of precision and 82 % of recall. Xerces-J is one of the largest systems that we studied with good number of changes applied over 10 years and our proposal successes to detect almost all of them with more than 96 % of precision and 88 % of recall. Thus, overall we can conclude that using our approach we could identify most of the applied operations correctly with an average of 85 % of precision and recall. The industrial system MROI-Ford was extensively modified during more than 10 years leading to more than 331 operations to detect. Our approach was successful to detect most of them with a precision of 96 % and recall of 95 %. Another interesting observation is that our technique does not have a bias towards the types of refactoring since most of them were detected.

We considered two other metrics related to automatic and manual correctness. Figure 13 shows that the suggested solutions using NSGA-II are similar to the "reference" solution provided manually by developers with more 78 % for all the 7 systems. However, it is a fastidious process for developers to manually merge parallel operations, thus sometimes better solutions can be provided by our automated change detection algorithm. A deviation with the set of reference solutions does not necessarily mean that there is an error/conflict with our multi-objective solutions, but it could be another possible good change detection solution different from the reference ones. To this end, we evaluated the suggested solutions by NSGA-II manually and we calculated a manual correctness score.

With regards to manual correctness (MC), the precision and recall scores for all the models were improved since we found interesting refactoring alternatives that deviates from the reference ones proposed by the experts. For instance, the precision for the GMF graph model was improved from 77 % to 91 %. In addition, we found that a sequence of applying the refactoring is sometimes different between generated refactoring and reference ones. We found that sometimes a different sequence can reduce the number of refactoring used to detect changes. All these results were statistically significant on 51 independent runs using the Friedman test with a 95 % confidence level ($\alpha < 5$ %). We have also found the following results of the Vargha Delaney A_{12} statistic : a) On small and medium scale software models (GMF Map, GMF Graph, GanttProject, GMF Gen and JHotdraw) NSGA-II is better than all the other algorithms based on all the performance metrics with an A effect size higher than 0.92; b) On large scale software models (Xerces-J, ApacheAnt and MROI-Ford), NSGA-II is better than all the other algorithms with a an A effect size higher than 0.86.

4.5.3 Results for RQ3

In this section, we compare our NSGA-II adaptation to the current, state-of-the-art model changes detection approaches. To answer RQ3.1, we compared NSGA-II to another widely used multi-objective algorithm, MOPSO, using the same adapted fitness function as described

in Table 4. A more qualitative evaluation is presented in Fig. 11 illustrating the box plots obtained for the multi-objective metrics on the different projects. It is clear from the results of Fig. 11 that, for both the HV and IGD metrics, NSGA-II has boxplots with higher variability (difference between min and max HV values) on GMD-Gen and Xerces-J when compared to MOPSO. For GMF Map, Apache Ant, JHotDraw, and MROI-Ford systems, the box-plots for NSGA-II and MOSPSO have almost the same variability. This fact confirms the effectiveness of NSGA-II over MOPSO in finding a well-converged and well-diversified set of Pareto-optimal model changes detection solutions.

Next, we use all five metrics NR, AC-P, AC-R, MC and ICT to compare different model changes detection techniques: our NSGA-II adaptation, MOPSO, a mono-objective genetic algorithm (GA-Agg) that has a single fitness function aggregating the two objectives, mono-objective model changes detection (Ben Fadhel et al. 2012), Langer et al. (Langer et al. 2012) and the UMLDiff tool (Xing and Stroulia 2005). In order to make meaningful comparisons, we select the best solution for NSGA-II and MOPSO using a knee point strategy (Zitzler et al. 2003). Thus, for NSGA-II and MOPSO, we select the knee point from the Pareto approximation having the median IHV value. The results of the comparison from 51 runs are depicted in Table 4 and 5, and Figs. 12, 13, 14 and 15. It can be seen that both NSGA-II and MOPSO provide a better trade-off between our two objectives than a mono-objective algorithm (GA-Agg and (Ben Fadhel et al. 2012)) in all the systems.

As described in Fig. 12, for NR, the number of refactorings using NSGA-II is lower than MOPSO in all systems. The mono-objective GA approach that aggregates the two objectives in one provides the highest number of refactorings than NSGA-II, MOPSO, (Ben Fadhel et al. 2012) and the non-search-based tools in all the cases. This confirms that it is difficult to find trade-offs between conflicting objectives aggregated in only one fitness function. The GR, Langer et al. (Langer et al. 2012; Ben Fadhel et al. 2012) and the UMLDiff tool (Xing and Stroulia 2005) provides the highest number of refactorings since they did not consider the minimization of the solutions size.

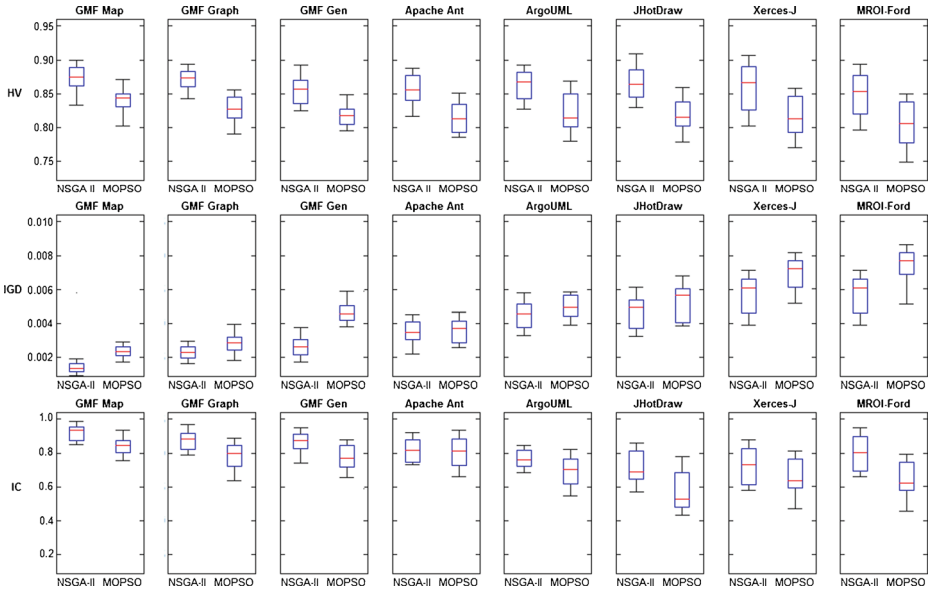

Fig. 11 Boxplots using the quality measures (**a**) HV, (**b**) IGD, and (**c**) IC applied to NSGA-II and MOPSO

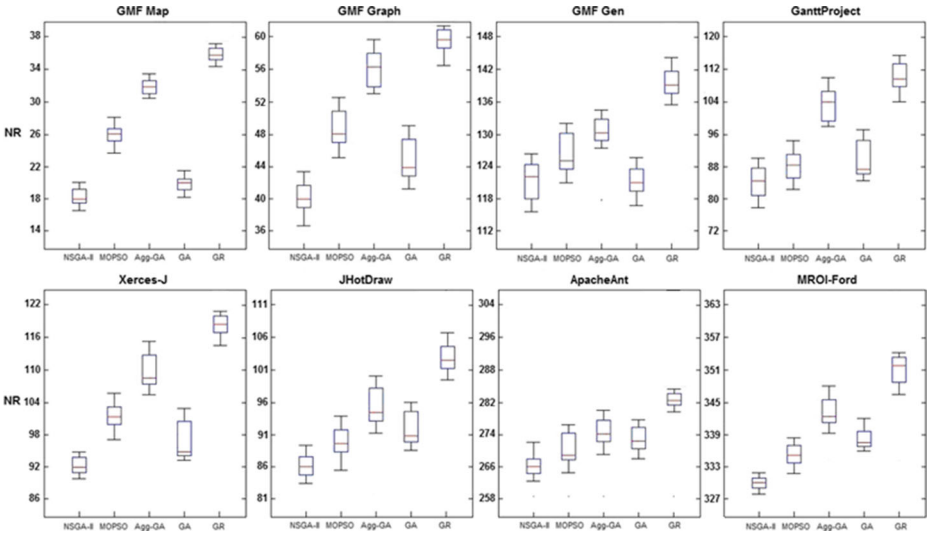

Fig. 12 Boxplots using the measure number of refactoring NR applied to NSGA-II, MOPSO, Agg-GA, GA (Ben Fadhel et al. 2012) and the simple deterministic greedy algorithm (GR) on the different systems for 51 independent runs on the different systems

For AC and MC, Figs. 13, 14 and 15 show that the solutions provided by NSGA-II have the highest manual and automatic correctness values on most of the systems. In fact, the average AC value for NSGA-II is 87 % and it is lower than 81 % for all the remaining algorithms on all the eight systems. The same observation is valid for MC, NSGA-II has the highest MC average value with 90 % while the remaining algorithms their MC average is lower than 84 %. Figures 8 and 9 reveal also an interesting observation that there is no correlation between the

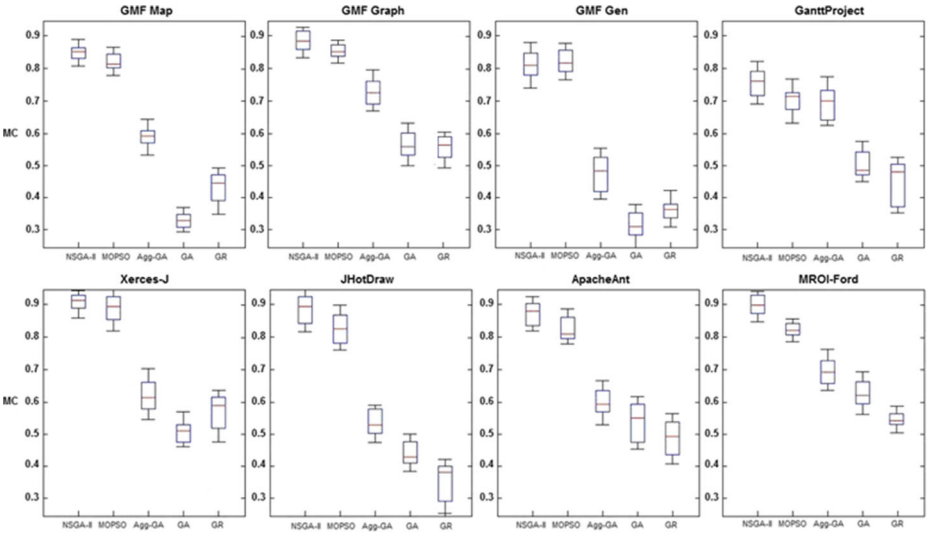

Fig. 13 Boxplots using the measure manual correctness MC applied to NSGA-II, MOPSO, Agg-GA, GA (Ben Fadhel et al. 2012) and the simple deterministic greedy algorithm (GR) on the different systems for 51 independent runs on the different systems

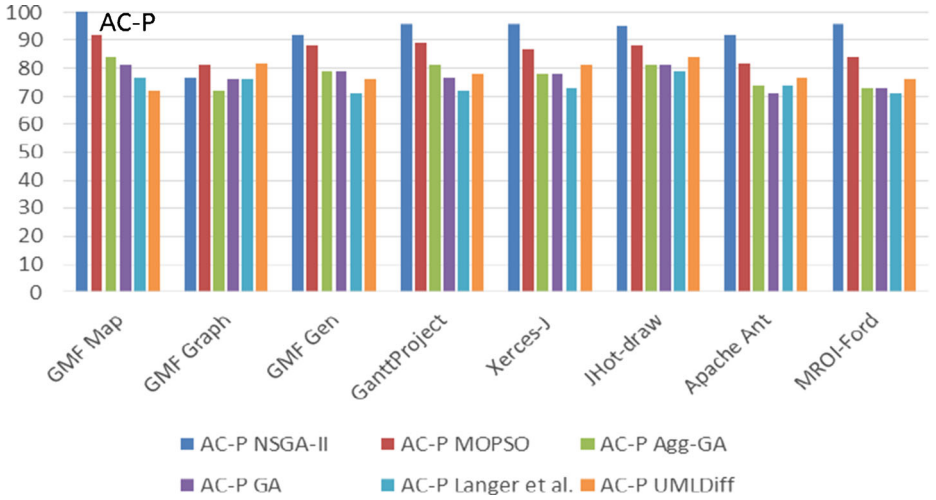

Fig. 14 Average automatic correctness precision of NSGA-II, MOPSO, Agg-GA, GA (Ben Fadhel et al. 2012), Langer et al., and UMLDiff on the different systems for 51 independent runs on the different systems

number of refactorings to detect and the correctness values. More precisely, we sort AC and MC based on the number of refactorings for each open source system. From this data, we conclude that AC and MC are not necessarily affected negatively by a larger number of refactorings to detect. For example, MC even increases from 88 % to 96 % when the number of refactorings increases from 122 to 341. Thus, we can conclude that our proposal shows a good scalability and is not affected negatively by the number of refactorings.

Figures 12, 13, 14 and 15 illustrate also the comparison between the results obtained using NSGA-II and two mono-objective genetic algorithms (Agg-GA and our previous work (Ben Fadhel et al. 2012)). It is clear enough that NSGA-II preforms better in terms of reducing the number of refactorings used to detect changes. In fact, in the eight systems NSGA-II generates lower number of refactoring than both mono-objective algorithms with a higher manual and

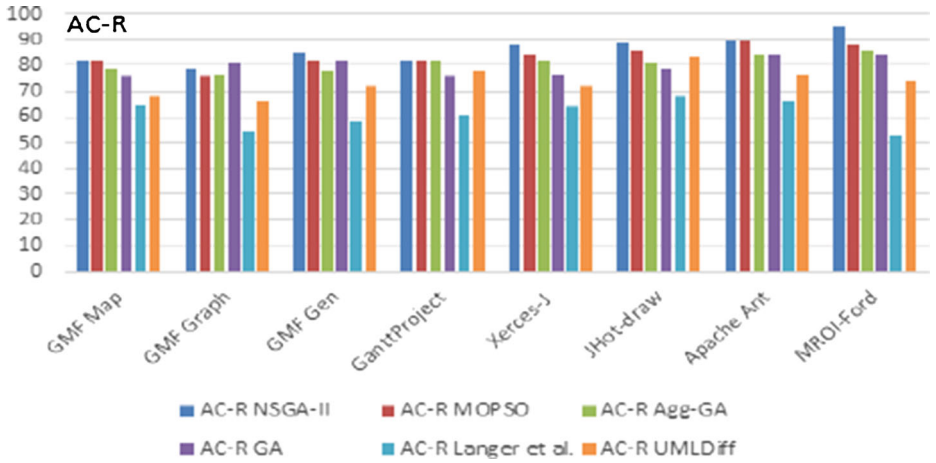

Fig. 15 Average automatic correctness recall of NSGA-II, MOPSO, Agg-GA, GA (Ben Fadhel et al. 2012), Langer et al., and UMLDiff on the different systems for 51 independent runs on the different systems

automatic correctness. In fact, reducing the number of refactoring improves the correctness of generated solution by eliminating overlaps, un-relevant and redundant refactorings. An example of un-relevant refactoring eliminated using NSGA-II is *move method(m, A, B)* then *move method(m, B, A)*. Overall, we obtained the same average score of correctness using NSGA-II and GA on the eight systems.

We also compared our NSGA-II detection results, described in Figs. 12, 13, 14 and 15, and the approach proposed by Langer et al. (Langer et al. 2012) and UMLDiff (Xing and Stroulia 2005) on the eight open source systems. We slightly improved the precision scores on all the eight systems using our approach. However, the recall of our proposal is much better than (Langer et al. 2012; Xing and Stroulia 2005). For instance, we improved the recall for GMF Graph from 54 to 79 %. When we analyzed manually the results, we found that composite refactoring operation, such as *Specialize Reference Type* and *Pull Up Feature* were not detected at all by both approaches (Langer et al. and UMLDiff) and we detected them using our approach. In the context of this experiment, we can conclude that our technique was able to identify model changes, in average, more accurately than (Langer et al. 2012) and UMLDiff (Xing and Stroulia 2005). In addition, we reduced the number of refactoring used to detect changes using NSGA-II as described in Fig. 12. All these results were statistically significant on 51 independent runs using the Friedman test with a 95 % confidence level ($\alpha < 5$ %).

It is difficult to predict in general that a specific search algorithm is the best for a specific problem since the only valid proof is the experimentation results. Thus, we compared the results of NSGA-II with another multi-objective search algorithm, namely multi-objective particle swarm optimization (MOPSO). The detection results for MOPSO were also acceptable and very close to NSGA-II performance. In fact, this can be explained by two main reasons. First, the problem adaptation is the same for both algorithms (solution representation, change operators and fitness function). Second, the two algorithms have common parts and the main difference between them is the crowding distance and change operators.

The execution time of NSGA-II is invariably lower than that of MOPSO with the same number of iterations (100,000) as described in Fig. 16, however the execution time required by Mono-objective algorithms (32 minutes in average) is lower than both NSGA-II and MOPSO (48 minutes in average for NSGA-II and 52 minutes for MOPSO). It is well known that a

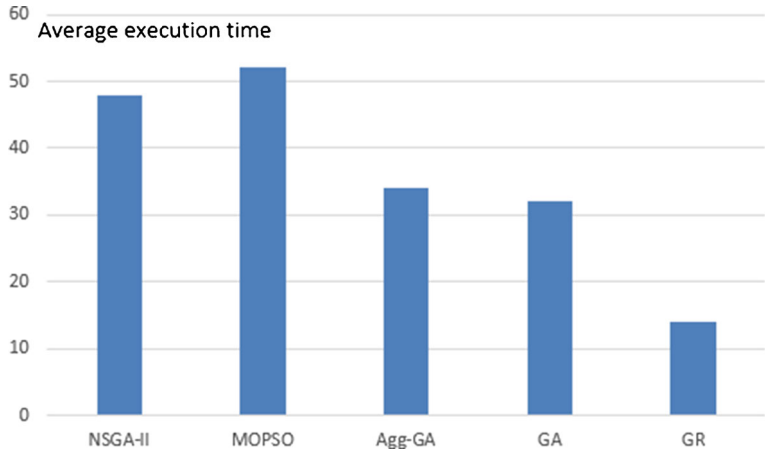

Fig. 16 Average execution time (in minutes) of NSGA-II, MOPSO, Agg-GA, and GA (Ben Fadhel et al. 2012) on the different systems for 51 independent runs on the different systems

mono-objective algorithm requires less execution time for convergence since only one objective is handled. However, the execution times of NSGA-II and MOPSO are not so far from those of mono-objective algorithms. For this reason, we can say that the difference in running time is more than compensated by the benefits provided by NSGA-II in terms of accurate detection of model changes.

In conclusion, we answer RQ3, the results support the claim that our NSGA-II formulation provides a good trade-off between both objectives, and outperforms on average the state-of-the-art of model changes approaches, both search-based and non-search-based ones.

4.5.4 Results for RQ4

We asked all the subjects of our experiments to take a study questionnaire to collect their opinions about our model changes detection tool and its relevance. Subjects were first asked to fill out a pre-study questionnaire containing five questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, their familiarity with changes detection and software refactoring and evolution tools. Each software engineer who accepted an invitation to participate in the study, received a questionnaire, a manuscript guide that helps to fill the questionnaire, an executable version of our tool and a brief documentation about the tool, and the required inputs to execute our tool from the studied systems in our experiments.

In the first part, the post-study questionnaire asked participants to rate their agreement on a Likert scale from 1 (complete disagreement) to 5 (complete agreement) with the following statements: 1. The proposed multi-objective model changes detection is a desirable feature in integrated development environments. 2. The detection of changes at the model level is a useful way to help developers understanding the evolution of a software and performing everyday design, implementation and maintenance activities. In the second part of the questionnaire, we asked the subjects to specify the possible usefulness to perform some activities such as quality assurance/assessment, effort prediction, model inspection, and features update and integration. In the third part, we asked the programmers about possible improvements of our multi-objective tool.

The questionnaire is completed anonymously thus ensuring confidentiality and this study were approved by the IRB at the University of Michigan: “Research involving the collection or study of existing data, documents, records, pathological specimens, or diagnostic specimens, if these sources are publicly available or if the information is recorded by the investigator in such a manner that participants cannot be identified, directly or through identifiers linked to the participants”. During the entire process, subjects were encouraged to think aloud and to share their opinions, issues, detailed explanations and ideas with the organizers of the study (one graduate student and one faculty from the University of Michigan) and not only answering the questions. A brief tutorial session was organized for every participant around refactoring and the used detection tool to make sure that all of them have a minimum background to participate in the study. In addition, all the developers performed the experiments in a similar environment: similar configuration of the computers, tools (Eclipse, Excel, etc.) and facilitators of the study. Because some support was needed for the installation of the changes detection techniques considered in our experiments, we added a short description of this instruction for the participants.

As described in Fig. 17, the agreement of the participants was 4.1 and 4.4 for the first and second statements respectively. This confirms the usefulness of our approach for the software

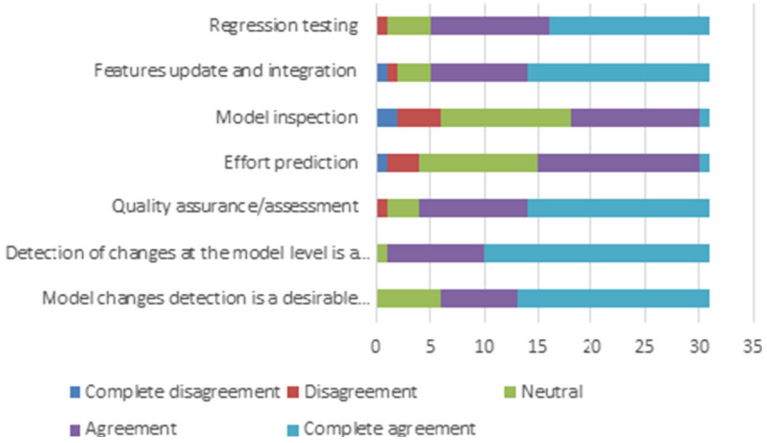

Fig. 17 The results of the study questionnaire

developers considered in our experiments. Regarding the possible usefulness to perform some activities, the developers agreed that quality assurance/assessment, regression testing and features understanding update are the three main activities where the detection of changes could be very helpful with an agreement of 4.1. The two other activities of effort prediction and model/code inspection are considered less relevant for our tool with an agreement of 3.7. In fact, most of the software engineers we interviewed found that the detected model changes give relevant quick advices about possible refactoring opportunities to improve the quality, what features were modified, localize errors/bugs and what are the test cases that should be executed to cover the new changes.

The remaining questions of the post-study questionnaire were about the benefits and also limitations (possible improvements) of our interactive approach. We summarize in the following the feedback of the developers. They found that the detection of changes at the model level process is much more helpful than the traditional analysis of changes at the code level to find refactoring opportunities and understand the evolution of the system. They consider the use of traditional code changes as a time consuming process, and it is easier to interpret the results of detected model changes for some tasks. Most of the participants mention that our automated approach to detect model changes is faster than manual exploration of the commits data since they spent a long time using regular version control tools to find the locations of the desired changes. The developers also considered that the detection of changes at the model level may provide a useful higher level of abstraction since the detected changes represent a summary of those detected at the code level. In fact, four of the interviewed developers mention that the code changes are very low level comparing to the model changes leading to a useful summary of the system evolution. In addition, seven out of the interviewed developers mention that they liked the high-level description of the changes (complex refactorings) comparing to the atomic changes described by regular control version tools. Another important feature that the participants mention is that our approach allows them to take the advantages of using multi-objective optimization for model changes detection without the need to learn anything about optimization and exploring the Pareto front to select one “ideal” solution based on their preferences.

The participants also suggested some possible improvements to our multi-objective approach. Some participants believe that it will be very helpful to extend the tool by adding a new feature is to use some visualization techniques to better present the results of the detected

changes. The developers also proposed to explore the area of impact changes analysis at the model level as a complementary step of our technique after the detection of model changes. The participant also suggested to improve the tool by including the ability to navigate between the model and code levels in an efficient way to check the changes at the same time at both the model and code levels.

We discuss also, in the following, how the shape of the Pareto front can help developers to select the best model changes solution based on their preferences.

Figure 17 depicts the different Pareto surfaces obtained on two open source systems and one industrial project (GMFGraph, JHotDraw and MROI-Ford) using NSGA-II to optimize both objectives related to minimizing the number of refactorings and maximizing the correctness (similarity to the expected version). Due to space limitations, we show only some examples of the Pareto-optimal front approximations obtained which differ significantly in terms of size. Similar observations 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 two objectives based on their own preferences. Based on the plots of Fig. 18, the software developer could decrease the number of detected refactorings while controlling visually the correctness. In this way, the developer can select the preferred model changes detection solution to realize.

When analyzing the results manually, we found that the missing 10 % of correctness is related to slight differences with the expected model which cannot be detected since they have to be formulated using atomic changes and not refactorings. In the worst case, the number of missed changes did not exceed 10. The selected solution at the knee point is just for the proof-of-concept of the validity of our approach but the programmers may select a different solution based on their preferences to give different weights to the objectives when selecting the best one.

One striking feature about all the three plots is that the number of refactorings increases slowly with a fast increase in the correctness of the solutions up to the knee point, marked in each figure. It is very interesting to note that this property of the Pareto-optimal front is apparent in all the problems considered in this study. It is likely that a software engineer would be drawn to this knee point as the probable best trade-off between our both objectives. Without any consideration of the number of refactorings in the search process, one would obtain the highest correctness in solution all the time, but Fig. 17 shows how a better changes detection solution in correctness can be obtained by sacrificing just a little in the number of refactorings. Another interesting observation in Fig. 17 is that the number of solutions in the Pareto fronts is not high, thus this can help the developers to select the best solution based on their preferences.

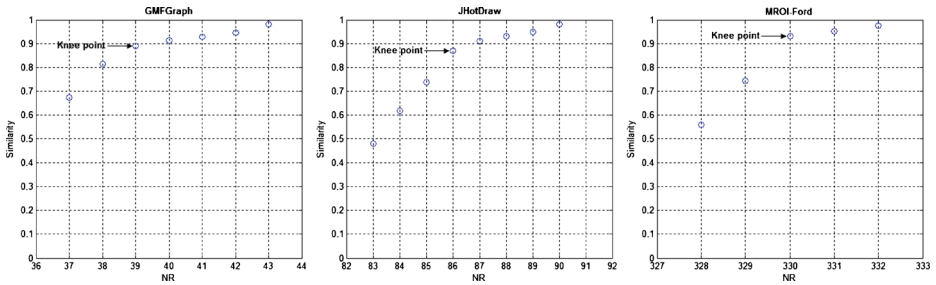

Fig. 18 Pareto fronts for NSGA-II obtained on GMF Graph (*small*), JHotDraw (*medium*), and MROI-Ford (*large*)

We also compared the model changes detection solution at the knee-point for MROI-Ford with the best solution that maximizes only the correctness to understand why the former solution finds a good trade-off. We found that the knee-point solution with lower number of refactorings did not include any redundancy and most of the refactorings are complex ones (not atomic). Hence we conclude that RQ4 is affirmed and that the multi-objective approach has value for software engineers in a real-world setting that can help programmers to understand detected changes while maximizing the correctness. In fact, all the interviewed developers mention that they could easily understand the solution with composite changes comparing to atomic ones.

5 Threats to Validity

Following the methodology proposed by Wohlin et al. (Wohlin et al. 2012), there are four types of threats that can affect the validity of our experiments. We consider each of these in the following paragraphs.

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. We used the Friedman test with a 95 % confidence level to test if significant differences existed between the measurements for different treatments. This test makes no assumption that the data is normally distributed and is suitable for ordinal data, so we can be confident that the statistical relationships we observed are significant. Another threat related to our approach is that some good solutions can be penalized not because that all refactorings were detected but due to few missed atomic changes. To address this threat, we may extend our work to combine the detection of refactorings with atomic changes as described in the conclusion section. In addition, our approach just focuses on the use of information available just at the model level, however some complex scenarios of changes may also require to consider the code level to improve the precision of our results. In addition, our multi-objective formulation treats the different types of refactoring with the same weight in terms of complexity. However, some refactoring types can be more complex than others to understand by developers.

Internal validity is concerned with the causal relationship between the treatment and the outcome. We dealt with internal threats to validity by performing 51 independent simulation runs for each problem instance. This makes it highly unlikely that the observed results were caused by anything other than the applied multi-objective model changes approach. Another threat is related to our choice of averaging rather than intelligently weighting the single objective function for the mono-objective genetic algorithm. Further experiments are required to evaluate the performance of the mono-objective approach with different weights to each component of the fitness function.

Construct validity is concerned with the relationship between theory and what is observed. To evaluate the results of our approach, we selected solution at the knee point as a proof-of-concept of the validity of our approach, but the programmers may select a different solution based on their preferences to give different weights to the objectives when selecting the best changes detection solution. Most of what we measure in our experiments are standard metrics such as IHV, IGD, etc. that are widely accepted as good proxies for quality. The different developers involved in our experiments may have sometimes divergent opinions about the detect changes in terms of correctness and readability. We considered in our experiments the majority of votes from the developers.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on seven different widely used open-source systems belonging to different domains and having different sizes, and one industrial project. 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 our findings. In addition, the comparison of the performance of NSGA-II to other multi-objective algorithms is limited to MOPSO. Further empirical studies are required to deeply evaluate the performance of NSGA-II using the same problem formulation.

6 Related Work

With respect to the contribution of this work, we present three lines of related work. First, we survey existing work for detecting changes on program code. Second, we elaborate on dedicated approaches for detecting changes for models which clearly represent the most closely related approaches to the presented approach. Finally, we also state further applications of search-based techniques in the field of software engineering.

6.1 Change Detection for Program Code

The easiest way to capture applied refactorings on program code is to track their execution in the development environment directly. Refactoring tracking is realized by (Dig et al. 2008; Ekman and Asklund 2004; Robbes 2007) in programming environments. All these approaches highly depend on the used environment which has to track the applied refactorings. Furthermore, manually performed refactorings are not detectable and refactorings which have been made obsolete by successive changes might be wrongly indicated. However, one major advantage is that the problem of hidden changes is not occurring, because every change is tracked.

In contrast to change tracking approaches, state-based refactoring detection approaches aim to reveal refactorings a posteriori on the base of the two successively modified versions of a software artifact. The detection of atomic changes on program code has a long history in computer science as pointed out by (Fluri et al. 2007), but is still an ongoing research topic (Maoz et al. 2010). Besides these approaches, several approaches are tailored to detect refactorings in program code. For instance, Dig et al. (Dig et al. 2006) propose an approach to detect applied refactorings in Java code. They first perform a fast syntactic analysis, and subsequently, a semantic analysis in which also operational aspects like method call graphs are considered. A similar approach is followed by (Weissgerber and Diehl 2006). After a preprocessing and a syntactical analysis have been conducted, conditions indicating the application of a refactoring are evaluated. In (Kim et al. 2012; Prete et al. 2010), a very recent approach for detecting refactorings improving several open issues of previous approaches has been proposed. In particular, the *REF-FINDER* tool is presented which is capable of detecting complex refactorings, which comprise a set of atomic refactorings using logic-based rules executed by a logic programming engine. However, refactorings which may overlap with atomic changes, referred to as *floss refactorings*, are only mentioned as future work. Besides these syntax-oriented approaches, a heuristic-based approach is presented in (Demeyer et al. 2000) in which a combination of various software measures as indicator for an application of a certain refactoring is used. For instance, a decrease in a method's size, among other measures, is used to indicate that the refactoring *Split Method* has been applied.

6.2 Change Detection for Models

Similar to change tracking for programming environments, the usage of change tracking in model environments has been proposed in (Koegel et al. 2010) coming with the same advantages and disadvantages as discussed before for program code. The state-based comparison of models attracted much attention in the last years that led to a huge list of different model comparison approaches which has been surveyed in (Cicchetti and Di Ruscio 2007). In (Kehrer et al. 2012), the main aspects of model comparison have been identified being difference calculation, representation, and finally, visualization. The content of this paper fits into the first aspect.

Most approaches for model comparison apply a two step-process: they first identify the correspondences between model elements of two different model versions, and second, based on these correspondences, the differences between the two model versions are derived. In the context of software evolution, difference calculation has been investigated intensively as witnessed by a number of approaches ranging from simple text comparisons based on the XMI serializations of models to dedicated model differencing techniques. Table 6 depicts a summary of the approaches discussed in the following and illustrates a comparison of them with the approach presented in this paper (cf. section 2 of Table 5). To give a comprehensive

Table 6 Comparison of selected comparison approaches for programs, models, and ontologies

	Language-specific	Composite changes		Hidden changes	
		Detectable	Additional rules	Detectable	Additional rules
REF-FINDER	yes	yes	yes	no	n.a.
Dig et al.	yes	yes	yes	no	n.a.
Weissgerber et al.	yes	yes	yes	no	n.a.
Demeyer et al.	yes	yes*	yes	yes*	no
EMFCompare	no	no	n. a.	n.a.	n.a.
Alanen & Porres	no	no	n.a.	n.a.	n.a.
DSMDiff	no	no	n.a.	n.a.	n.a.
SiDiff	no	no	n.a.	n.a.	n.a.
Barret et al.	no	no	n.a.	n.a.	n.a.
Riviera & Vallecillo	no	no	n.a.	n.a.	n.a.
SiLift	no	yes	no	no	n.a.
ETL	no	yes	yes	no	n.a.
UMLDiff	yes	yes	yes	no	n.a.
Küster et al.	yes	yes	yes	no	n.a.
Vermolen et al.	yes	yes	yes	yes°	yes
Langer et al.	no	yes	no	yes°	no
Hartung & Rahm	yes	yes	yes	no	n.a.
GenDiff	no	yes	no	yes	no

Legend:

* an estimation possible refactorings is reported

° hidden operations are only partially supported

overview, we also list approaches for detecting changes in program code and ontologies in this table (cf. section 1 and section 3 of Table 5, respectively).

A specialized differencing method has been introduced to compare UML models by Xing and Stroulia (Xing and Stroulia 2005). Another UML specific differencing approach is proposed by Nejati et al. (Nejati et al. 2007), which is specifically tailored for matching UML state machines. SiDiff (http://pi.informatik.uni-siegen.de/Projekte/sidiff) started also as a comparison tool for UML models but has been further generalized to detect changes for arbitrary models by having an internal generic graph representation to which rarity languages can be mapped. Alanen and Porres (Alanen and Porres 2003) discussed how to detect differences between MOF-based models. Rivera and Vallecillo (Rivera and Vallecillo 2008) propose to compare model based on identifier as well as structural similarities by using rules defined in Maude. Thus, before the models are compared, they are translated to corresponding Maude representations on which the comparison is actually performed. DSMDiff (Lin et al. 2007) is another protagonist for computing differences between models, irrespectively of their metamodel. Finally, EMF Compare (Brun and Pierantonio 2008) is an Eclipse plug-in for comparing EMF-based models providing out-of-the-box matching and differencing support as well as several extension points to develop tailored matching and differencing support. All the approaches mentioned in this paragraph derive only atomic changes, namely additions, deletions, updates, and some of them, moves.

A few approaches use an additional third step in the model comparison process to combine atomic changes to composite changes by using dedicated aggregation rules. In particular, the output of the second phase of model comparison is used as input for the third phase, i.e., the before calculated atomic changes are rewritten to composite ones by applying the aggregation rules.

The starting point is the approach by Xing and Stroulia (Xing and Stroulia 2006) for detecting refactorings in evolving software models which is integrated in *UMLDiff*. For detecting refactorings, change pattern queries have to be developed for each refactoring. These change pattern queries are executed on a difference model obtained by a state-based model comparison to report refactoring applications. The approach is tailored to UML models and the detection of hidden changes is not mentioned.

In Langer et al. (Langer et al. 2012), we have presented an approach based on graph transformation rules to detect refactorings in software models. In contrast to (Xing and Stroulia 2006), no additional rules have to be developed for detecting changes. On the contrary, the rules for executing the operations are reused also for detecting applications of them. Using an iterative and incremental operation detection approach, overlapping operation sequences are supported as long as the postconditions of preceding operations that enable other succeeding operations are fulfilled in the revised model. However, operation sequences, in which the postconditions are not fulfilled in the revised model, are not supported. Thus, with the search-based approach presented in this paper, we are able to improve the results of (Langer et al. 2012) concerning the recall, as discussed in the evaluation section of this paper.

Kehrer et al. (Kehrer et al. 2011) follows a similar path by proposing to derive dedicated detection rules from graph transformation rules that represent composite operations. The derived detection rules are then matched on difference models containing the atomic operations that have been applied between two versions of a model. The approach by Vermolen et al. (Vermolen et al. 2011) copes with the detection of evolution steps between different versions of a metamodel to allow for a higher automation in model migration, i.e., to adapt the existing models to the new metamodel version. They use a difference model comprising atomic changes as input and calculate, on this basis, composite changes. The approach is tailored to the core of object-

oriented metamodeling languages, but follows a similar methodology as UMLDiff. However, the approach requires developing an additional detection rule for every possible change combination which represents a practical challenge if a larger set of refactoring operations is used.

In the area of business process modeling, there is the work of Küster et al. (Küster et al. 2008) for calculating hierarchical change logs for business process models including compound changes in the absence of recorded change logs. The authors apply the concept of Single-Entry-Single-Exit fragments to calculate the hierarchical change logs after computing the correspondences between two process models. Thereby, several atomic changes are hidden behind one compound change representing an introduction or deletion of a model fragment. However, changes that cross-cut the containment hierarchy is not considered.

In summary, the detection of composite operations and hidden operations on models is currently only discussed by Vermolen et al. (Vermolen et al. 2011) and Langer et al. (Langer et al. 2012). Nevertheless, when following (Vermolen et al. 2011) for detecting hidden refactorings, additional rules have to be developed by hand for each possible combination which makes the elaboration of the complete search space practically impossible due to the huge size of possible refactoring combinations and even more challenging due to eventually overlapping atomic changes. In (Langer et al. 2012), no additional rules are needed, but some refactorings are not detectable when the operation's postconditions are not valid in the final model version. In this paper, we propose an approach which is able to work without additional knowledge about refactoring sequences and use an efficient search technique which has been proven useful in our evaluations to detect both kinds of hidden operations. Furthermore, our approach also contributes a method that may be applied to detect floss refactorings on programs which is currently not supported by existing approaches working on the program code level, except the approach by Demeyer et al. who use a relaxed notion of refactorings' pre- and postconditions in terms of software metrics. Based on these metrics, their approach may provide a rather rough estimation of the applied refactorings, however, as pointed out in (Weissgerber and Diehl 2006), their relaxed notion has a negative impact on the precision, which seems to be rather low for some refactorings types.

6.3 Search-Based Software Engineering

Our approach is inspired by contributions in search-based software engineering (SBSE) (Harman and Jones 2001). As the name indicates, SBSE uses a search-based approach to solve optimization problems in software engineering. Once a software engineering task is framed as a search problem, by defining it in terms of solution representation, fitness function, and solution change operators, there are a multitude of search algorithms that can be applied to solve that problem. To the best of our knowledge, inspired among others by the roadmap paper of Harman (Kessentini et al. 2008; Harman et al. 2009), the idea of treating model changes detection as a combinatorial optimization problem to be solved by a search-based approach was not studied before our previous mono-objective work (Ben Fadhel et al. 2012).

The closest work in MDE to our contribution is our recent work (Kessentini et al. 2008) about the use of heuristic search techniques for model transformation. In fact, we proposed MOTOE (MOdel Transformation as Optimization by Example), an approach to automate model transformation using heuristic-based search. MOTOE uses a set of transformation examples to derive a target model from a source model. The transformation is seen as an optimization problem where different transformation possibilities are evaluated and a quality associated to each one depending on its conformance with the examples at hand. Several other works are proposed to find refactoring opportunities using search-based techniques (Harman and Tratt 2007; Seng et al.

2006). For example, in (Seng et al. 2006) defect detection is considered as an optimization problem. The authors use a combination of 12 metrics to measure the improvements achieved when sequences of simple refactorings are applied, such as moving methods between classes. The goal of the optimization is to determine the sequence that maximizes a function, which captures the variations of a set of metrics. Recently, Mel et al. (Ó Cinnéide M et al. 2012) evaluated the impact of refactoring on software quality metrics using search-based techniques.

7 Conclusion and Future Work

In this paper we introduced a new approach for model change detection based on multi-objective optimization techniques. Our algorithm starts by randomly generating a set of refactoring combinations, executing them on the initial model to generate a revised model, and then evaluates the quality of the proposed solution (refactorings) by comparing the generated revised model and the expected one. The goal is to maximize their similarity and at the same time to minimize the number of generated refactorings.

We have evaluated our approach on real-world models extracted from seven open source systems and one industrial project. The experimental results indicate that detected changes are comparable to those expected by the developers. We compared our technique with existing approaches and obtained better results.

Although our approach has been evaluated with real-world models with a reasonable number of changes to detect, we are working now on larger models and with larger lists of refactoring types. This is necessary to investigate more deeply the applicability of the approach in practice, but also to study the performance of our approach when dealing with very large models. We will be extending our evaluation on different industrial automotive projects in collaboration with several team managers. More generally, we plan to extend this work by classifying the detected changes based on a degree of risk and considering additional objectives to optimize during the detection process such the refactoring types to detect. Furthermore, we will further evaluate the performance of NSGA-II with several other multi-objective algorithms and existing model changes detection tools. In addition, we are planning to combine the detection of both atomic changes and refactorings in our approach by maximizing the number of refactorings and minimizing the number of atomic changes. So, most of the combinations of atomic changes that may lead to refactorings will be performed to maximize the first objective and the remaining atomic changes will correspond to added or deleted elements but not refactorings.

In our approach, some of the changes cannot be detected accurately with only information collected from the model level. It could be a very interesting future research directions of this paper to combine both code and model levels, when necessary, to detect some complex changes (like the example that you presented) such as comparing the size the methods at the code level to improve the accuracy of detecting changes. The advantage of our current approaches that it did not require the history of previous commit messages (not always available and accurate in real world settings, especially with old and legacy system projects) as input to generate detected refactorings. However, the use of commit messages, when available (and if we consider the code level), can be integrated to our approach as an additional objective to better detect changes. In our current multi-objective adaptation, we considered the refactorings with the same weight in terms of complexity. An interesting future work is to give different complexity weights to the refactoring types to prioritize solutions containing the lower number of complex refactorings while maximizing the correctness. Furthermore, some of the participants highlighted these

difficulties to identify and evaluate a change just based on the models and suggested to include a multi-level view feature to our detection tool when they can see both the model and code views. Thus, we are planning to extend our work to visualize changes for programmers at both the code and model levels based on their preferences in terms of the required degree of details.

References

Alanen M, Porres I (2003) Difference and union of models. Proc Int Conf Unified Model Language (UML'03) LNCS 2863:2–17, Springer

Arcuri A, Briand LC (2011) A practical guide for using statistical tests to assess randomized algorithms in software engineering. Proc 33rd Int Conf Software Eng (ICSE '11)

Bechikh S, Ben Said L, Ghédira K (2010) Estimating nadir point in multi-objective optimization using mobile reference points. Proc IEEE Congress Evolution Comput (CEC'10): 2129–2137

Ben Fadhel A, Kessentini M, Langer P, Wimmer M (2012) Search-based detection of high-level model changes. Proc 28th IEEE Int Conf Software Maintenance, Italy

Briand LC, Labiche Y, Soccar G (2002) "Automating impact analysis and regression test selection based on UML designs". Proc Int Conf Software Maintenance. IEEE: 251–260

Brosch P, Langer P, Seidl M, Wieland K, Wimmer M, Kappel G, Retschitzegger W, Schwinger W (2009) An example is worth a thousand words: composite operation modeling by-example. Proc MoDELS'09, Springer, 271–285

Brosch P, Seidl M, Wimmer M, Kappel G (2012) Conflict visualization for evolving UML models. J Object Technol 11(3):2, 1-30

Brown WJ, Malveau RC, Brown WH, Mowbray TJ (1998) "Anti patterns: refactoring software, architectures, and projects in crisis,". John Wiley & Sons, ISBN 978-0471197133

Brun C, Pierantonio A (2008) Model differences in the eclipse modeling framework. UPGRADE, Europ J Inform Prof 9(2):29–34

Cicchetti D, Di Ruscio A (2007) Pierantonio: a metamodel independent approach to difference representation. J Object Technol 6(9):165–185

Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans Evol Comput 6:182–197

Demeyer S, Ducasse S, Nierstrasz O (2000) Finding refactorings via change metrics. Proc Conf Object-Orient Prog, Syst, Languages, Appl (OOPSLA'00), ACM. 166–177

Dig D, Comertoglu C, Marinov D, Johnson R (2006) Automated detection of refactorings in evolving components. Proc Europ Conf Object-Orient Prog (ECOOP'06), vol. 4067 of LNCS, Springer 404–428

Dig D, Manzoor K, Johnson RE, Nguyen TN (2008) Effective software merging in the presence of object-oriented refactorings. IEEE Trans Softw Eng 34(3):321–335

Eiben AE, Smit SK (2011) Parameter tuning for configuring and analyzing evolutionary algorithms. Swarm Evol Comput 1(1):19–31

Ekman T, Asklund U (2004) Refactoring-aware versioning in eclipse. Electronic Notes Theor Comput Sci 107:57–69

Fluri B, Würsch M, Pinzger M, Gall H (2007) Change distilling: tree differencing for fine-grained source code change extraction. IEEE Trans Software Eng 33(11):725–743

Fonseca CM, Fleming PJ (1993) Genetic algorithms for multi-objective optimization: formulation, discussion and generalization. In: Forrest S (ed) Proceedings of the fifth international conference on genetic algorithms. Morgan Kauffman, San Mateo, pp 416–423

Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring – improving the design of existing code; 1st ed. Addison-Wesley

France R, Rumpe B (2007) Model-driven development of complex software: a research roadmap. Proc Int Conf Software Eng (ICSE'07): Future Software Eng; IEEE Comput Soc Press

Friedman M (1937) The use of ranks to avoid the assumption of normality implicit in the analysis of variance. J Am Stat Assoc (American Statistical Association) 32(200):675–701. doi:10.2307/2279372.JSTOR2279372

Gasevic D, Djuric D, Devedzic V (2009) Model driven engineering and ontology development; 2. edition, Springer

Gîrba T, Ducasse S (2006) Modeling history to analyze software evolution. J Softw Maint Evol Res Pract 18(3):207–236

Harman M, Jones BF (2001) Search-based software engineering. Inform Software Technol 43(14):833–839

Harman M, Tratt L (2007) Pareto optimal search based refactoring at the design level. Proc Genet Evolution Comput Conf (GECCO'07): 1106–1113

Harman M, Mansouri SA, Zhang Y (2009) Search based software engineering: a comprehensive analysis and review of trends techniques and applications. Technical report TR-09-03, King's College

T. Kehrer, U. Kelter, G. Taentzer (2011) A rule-based approach to the semantic lifting of model differences in the context of model versioning. Proc Int Conf Automated Software Eng (ASE'11), IEEE, 163–172
Kehrer T, Kelter U, Ohrndorf M, Sollbach T (2012) Understanding model evolution through semantically lifting model differences with SiLift. Proc 28th Int Conf Software Maintenance (ICSM'12), IEEE Comput Soc
Kessentini M, Sahraoui H, Boukadoum M (2008) Model transformation as an optimization problem. Proc MoDELS'08, vol. 5301 of Springer LNCS, 159–173
Kim M, Notkin D, Grossman DG Jr. (2012) Wilson: identifying and summarizing systematic code changes via rule inference. IEEE Trans Software Eng, early access article
Koegel M, Herrmannsdoerfer M, Li Y, Helming J, Joern D (2010) Comparing state- and operation-based change tracking on models. Proc IEEE Int EDOC Conf
Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge
Küster JM, Gerth C, Förster A, Engels G (2008) Detecting and resolving process modeling differences in the absence of a change log. Proc Int Conf Business Process Manag (BPM'08), LNCS, Springer, 244–260
Langer P, Wimmer M, Brosch P, Herrmannsdoerfer M, Seidl M, Wieland K, Kappel G (2012) A posteriori operation detection in evolving software models. J Syst Software
Li X (2003) A non-dominated sorting particle swarm optimizer for multiobjective optimization. Proc 2003 Int Conf Genet Evolution Comput (GECCO'03): 37–48
Lin Y, Gray J, Jouault F (2007) DSMDiff: a differentiation tool for domain-specific models. Eur J Inf Syst 16(4):349–361
Liu H, Yang L, Niu Z, Ma Z, Shao W (2009) Facilitating software refactoring with appropriate resolution order of bad smells. Proc ESEC/FSE'09: 265–268
Mansoor U, Kessentini M, Langer P, Wimmer M, Bechikh S, Deb K (2015) "MOMM: Multi-objective model merging". J Syst Software, TBD, TBD: 1–20
Mansoor U, Kessentini M, Langer P, Wimmer M, Bechikh S, Deb K (2015b) MOMM: multi-objective model merging. J Syst Softw 103:423–439
Maoz S, Ringert J, Rumpe B (2010) A manifesto for semantic model differencing. MoDELS workshops. LNCS 6627:194–203, Springer
Mens T, Tourwé T (2004) A survey of software refactoring. IEEE Trans Softw Eng 30(2):126–139
Moha N, Guéhéneuc Y-G, Duchien L, Meur A-FL (2009) DECOR: a method for the specification and detection of code and design smells. IEEE Trans Software Eng (TSE)
Nejati S, Sabetzadeh M, Chechik M, Easterbrook S, Zave P (2007) Matching and merging of statecharts specifications. Proc Int Conf Software Eng (ICSE 2007): 54–64. IEEE
O Cinnéide M (2001) Automated application of design patterns: a refactoring approach. Ph.D. Dissertation. Trinity College Dublin
Ó Cinnéide M, Tratt L, Harman M, Counsell S, Moghadam IH (2012) Experimental assessment of software metrics using automated refactoring. ESEM: 49–58
Opdyke WF (1992) Refactoring object-oriented frameworks. Ph.D. Dissertation. University of Illinois at Urbana-Champaign, Champaign, IL, USA. UMI Order No. GAX93-05645
Ouni A, Kessentini M, Sahraoui HA, Hamdi MS (2013) The use of development history in software refactoring using a multi-objective evolutionary algorithm. GECCO 1461–1468
Pilskalns O, Uyan G, Andrews A (2006) Regression testing uml designs. 22nd IEEE Int Conf Software Maintenance. ICSM'06. IEEE: 254–264)
Porres I (2003) Model refactorings as rule-based update transformations. Springer, Berlin Heidelberg, pp 159–174
Prete K, Rachatasumrit N, Sudan N, Kim M (2010) Template-based reconstruction of complex refactorings. Proc Int Conf Software Maintenance (ICSM'10), IEEE 1–10
Rivera J, Vallecillo A (2008) Representing and OPERATING WITH MODEL DIFFERENCES. Paige RF, Meyer B (eds.) TOOLS EUROPE 2008. LNBIP 11, 141–160. Springer
Robbes R (2007) Mining a change-based software repository. Proc Workshop Mining Software Repositories (MSR'07) IEEE Comput Soc: 15–23
Roberts DB, Johnson R (1999) Practical analysis for refactoring. Ph.D. Dissertation
Ruiz R, Stützle T (2007) A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. Eur J Oper Res 177(3):2033–2049
Sahin D, Kessentini M, Wimmer M, Deb K (2015) Model transformation testing: a bi-level search-based software engineering approach. J Software: Evol Process 27(11):821–837
Seng O, Stammel J, Burkhart D (2006) Search-based determination of refactorings for improving the class structure of object-oriented systems. Proc Genet Evolution Comput Conf (GECCO'06): 1909–1916
Vermolen S, Wachsmuth G, Visser E (2011) Reconstructing complex metamodel evolution; Tech. Rep. TUD-SERG-2011-026, Delft University of Technology
Weissgerber P, Diehl S (2006) Identifying refactorings from source-code changes. Proc Int Conf Automated Software Eng (ASE'06), IEEE 231–240

Wieland K, Langer P, Seidl M, Wimmer M, Kappel G (2013) Turning conflicts into collaboration. Comput Supported Coop Work 22(2-3):181–240
Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in Software Engineering. Springer
Xing Z, Stroulia E (2005) UMLDiff: An algorithm for object-oriented design differencing. Proc Int Conf Automated Software Eng (ASE 2005): 54–65. ACM
Xing Z, Stroulia E (2006) Refactoring detection based on UMLDiff change-facts queries. Proc 13th Work Conf Reverse Eng (WCRE'06), IEEE:263–274
Zitzler E, Thiele L, Laumanns M, Fonseca CM, da Fonseca VG (2003) Performance assessment of multiobjective optimizers: an analysis and review. IEEE Trans Evol Comput 7(2):117–132

Dr. 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 computational search techniques to software engineering (search-based software engineering), refactoring, software testing, model-driven engineering, software quality, and reengineering. 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. He is the general chair of SSBSE2016.

Usman Mansoor is currently a PhD 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.

Manuel Wimmer is a postdoctoral researcher in the Business Informatics Group (BIG). He received his Ms and PhD degrees in business informatics from the Vienna University of Technology in 2005 and 2008, respectively. Furthermore, he is one of the prize winners of the Award of Excellence 2008 assigned by the Austrian Federal Ministry of Science and Research for his PhD thesis.

In 2011/2012 he was on leave as research associate at the University of Málaga (working with Prof. Antonio Vallecillo). He was funded by an Erwin Schrödinger scholarship granted by the Austrian FWF (Fonds zur Förderung der Wissenschaftlichen Forschung).

His current research interests comprise Web engineering, model engineering, and reverse 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.

Ali Ouni has a PhD in computer science under the supervision of Pr. Marouane Kessentini (University of Michigan) and Pr. Houari Sahraoui (University of Montreal). He is now a research assistant professor at Osaka University, Japan. He is also a member of the SBSE research laboratory, University of Michigan, USA and member of the GEODES software engineering laboratory, University of Montreal, Canada. He received his

bachelor degree (BSc.) and his Master Degree Diploma (MSc.) in computer science from the Higher Institute of Applied Sciences and Technology (ISSAT), University of Sousse, Tunisia, respectively in 2008, and 2010. His research interests include the application of artificial intelligence techniques to software engineering (search-based software engineering). Since 2011, he published several papers in well-ranked journals and conferences. He serves as program committee member in several conferences and journals such as GECCO'14, CMSEBA'14, and NasBASE'15.

Dr. 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.