Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2014; **26**:692–713
Published online 12 February 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1644

Model refactoring using examples: a search-based approach

Adnane Ghannem[1,*,†], Ghizlane El Boussaidi[1] and Marouane Kessentini[2]

[1]*Software Engineering and IT Department, École de Technologie Supérieure, Montreal, QC, Canada*
[2]*CIS department, SBSE Research Lab, University of Michigan, Dearborn, MI, USA*

ABSTRACT

One of the important challenges in model-driven engineering is how to improve the quality of the models' design in order to help designers understand them. Refactoring represents an efficient technique to improve the quality of a design while preserving its behavior. Most of existing work on model refactoring relies on declarative rules to detect refactoring opportunities and to apply the appropriate refactorings. However, a complete specification of refactoring opportunities requires a huge number of rules. In this paper, we consider the refactoring mechanism as a combinatorial optimization problem where the goal is to find good refactoring suggestions starting from a small set of refactoring examples applied to similar contexts. Our approach, named model refactoring by example, takes as input an initial model to refactor, a set of structural metrics calculated on both initial model and models in the base of examples, and a base of refactoring examples extracted from different software systems and generates as output a sequence of refactorings. A solution is defined as a combination of refactoring operations that should maximize as much as possible the structural similarity based on metrics between the initial model and the models in the base of examples. A heuristic method is used to explore the space of possible refactoring solutions. To this end, we used and adapted a genetic algorithm as a global heuristic search. The validation results on different systems of real-world models taken from open-source projects confirm the effectiveness of our approach. Copyright © 2014 John Wiley & Sons, Ltd.

Received 27 February 2013; Revised 10 December 2013; Accepted 6 January 2014

KEY WORDS: software maintenance; model evolution; model refactoring; refactoring by example; heuristic method; genetic algorithm

1. INTRODUCTION

To cope with the changing and growing business needs, software systems are constantly evolving. Software evolution activities can span from maintenance to an entire replacement of the system [1]. Software maintenance is considered the most expensive activity in the software system life cycle [2]. According to the ISO/IEC 14764 standard, the maintenance process includes the necessary tasks to modify existing software while preserving its integrity [3]. Maintenance tasks can be seen as incremental modifications to a software system that aim to add or adjust some functionality or to correct some design flaws and fix some bugs. However, as time goes by, the system's conceptual integrity erodes [1], and its quality degrades; this deterioration is known in the literature as the software decay problem [4]. Therefore, maintenance tasks become more complex and costly.

A common and widely used technique to cope with this problem is to continuously restructure the software system to improve its structure and design. The process of restructuring object-oriented systems is commonly called refactoring [5]. According to Fowler [4], refactoring is the disciplined process of cleaning up code to improve the software structure while preserving its external behavior.

*Correspondence to: Adnane Ghannem, École de Technologie Supérieure, Montreal, QC, Canada.
†E-mail: adnane.ghannem.1@ens.etsmtl.ca

Automating refactoring operations necessarily helps coping with software complexity and keeping the maintenance costs from increasing. Many researchers have been working on providing support for refactoring operations (e.g., [6, 4, 7]). Existing tools provide different environments to manually or automatically apply refactoring operations to correct, for example, code smells [8]. Indeed, existing work has, for the most part, focused on refactorings at the source code level. Very few approaches tackled the refactoring process at the model level (e.g., [9–11]). Nevertheless, models are primary artifacts within the model-driven engineering (MDE) approach, which has emerged as a promising approach to manage software systems' complexity and specify domain concepts effectively [12]. In MDE, abstract models are refined and successively transformed into more concrete models including executable source code. The evolution of models and the transformations that manipulate them is crucial to MDE approaches; however, the maintenance process is still focused on source code.

Actually, the rise of MDE increased the interest and the need for tools supporting refactoring at the model level. Indeed, such a tool may be of great value for novice designers as well as experienced ones when refactoring existing models. However, there are many open and challenging issues that we must address when building such a tool. Mens and Tourwé [10] argue that most of the refactoring tools offer a semi-automatic support because part of the necessary knowledge for performing the refactoring remains implicit in designers' heads. Indeed, recognizing opportunities of model refactoring remains a challenging issue that is related to the model marking process within the context of MDE, which is a notoriously difficult problem that requires design knowledge and expertise [13]. Finding refactoring opportunities in source code has relied, for the most part, on quality metrics (e.g., [14–16]). However, some of these metrics (e.g., number of lines of code) and refactorings (e.g., removing duplicate code) do not apply at the model level. Hence, the designer needs to identify the useful and applicable metrics for a given model of the system and decide how to correctly combine these metrics to detect and propose a refactoring. In addition, existing work on refactoring relies on declarative rules to detect and correct defects (i.e., refactoring opportunities), and the number of types of these defects can be very large [17]. This problem's complexity is strongly increased when the designer is looking for an appropriate sequence of refactorings that corrects the entire set of the system's defects.

In this paper, we hypothesize that the knowledge required to propose appropriate refactorings for a given object-oriented model may be inferred from other existing models' refactorings when there is some similarities between these models and the given model. We propose model refactoring by example (MOREX), an approach to automate model refactoring using heuristic-based search. MOREX relies on a set of refactoring examples to propose sequences of refactorings that can be applied on a given object-oriented model. The refactoring is seen as an optimization problem where different sequences of refactorings are evaluated depending on the similarity between the model under analysis and the refactored models in the examples at hand. Our approach takes as input an initial model that we want to refactor, a base of examples of refactored models, and a list of metrics calculated on both the initial model and the models in the base of examples, and it generates as output a solution to the refactoring problem. In this case, a solution is defined as a sequence of refactoring operations that should maximize as much as possible the similarity between the initial model and the models in the base of examples. Because of the very large number of possible solutions (i.e., refactoring combinations), a heuristic method is used instead of an enumerative one to explore the space of possible solutions. Because the search space is very large, we use and adapt a genetic algorithm (GA) as a global heuristic search.

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

- We introduce a new refactoring approach based on the use of examples. Our proposal does not require the user to define explicitly the defect types but only to have some refactoring examples, it does not require an expert to write detection or correction rules manually, and it combines detection and correction steps.
- We report the results of an evaluation of our approach; we used refactoring examples extracted from eight object-oriented open-source projects. We applied an eight-fold cross-validation procedure. For each fold, one open-source project is evaluated by using the remaining seven systems as bases of examples. The average values of precision and recall computed from 31 executions on each project are around 85%, which allows us to say that the obtained results are

DOI: 10.1002/smr

promising. The effectiveness of our approach is also assessed using a comparative study between our approach and two other approaches.

The paper is organized as follows. Section 2 is dedicated to the basic concepts. Section 3 presents the overall approach and the details of our adaptation of the GA to the model refactoring problem. Section 4 describes the implementation and the experimental setting. Section 5 presents and discusses the experimental results. Related works are discussed in Section 6, and we conclude and outline some future directions to our work in Section 7.

2. BASIC CONCEPTS

This section defines some relevant concepts to our proposal, including model refactorings, software metrics, and heuristic search.

2.1. *Model refactorings*

'Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure' [18]. Model refactoring is a controlled technique for improving the design (e.g., class diagrams) of an existing model. It involves applying a series of small refactoring operations to improve the model quality while preserving its behavior. Many refactorings were proposed and codified in the literature (see, e.g., [4]). In our approach, we considered a subset of the 72 refactorings defined in [4]; we considered only those refactorings that can be applied to class diagrams as an example of design models. Indeed, some of the refactorings in [4] may be applied on design models (e.g. Move_Method, Rename_method, Move_Field, Extract_Class, etc.) while others cannot be (e.g., Extract_Method, Inline_Method, Replace_Temp_With_Query, etc.). The refactoring configuration for the experiments of our approach reported here consisted of the 12 refactorings described in the succeeding text (Table I). The choice of these refactorings was mainly based on two factors: (1) they apply at the model level (i.e., we focused on class diagrams), and (2) they can be linked to a set of model metrics (i.e., metrics that are impacted when applying these refactorings). The considered metrics are presented in the following subsection.

2.2. *Quality metrics*

Quality metrics provide useful information that helps in assessing the level of conformance of a software system to a desired quality such as *evolvability* and *reusability* [19]. Metrics can also help in detecting some similarities between software systems. The most widely used metrics for class diagrams are the ones defined by Genero *et al*. [20]. In the context of our approach, we used the 11 metrics defined in [20] to which we have added a set of simple metrics (e.g., number of private methods in a class and number of public methods in a class) that we have defined for our needs. The metrics configuration for the experiments reported here consisted of the 16 quality metrics described in Table II. All this metrics are related to the class entity, which is the main entity in a class diagram. Some of these metrics represent statistical information (e.g., number of methods, attributes, etc.), and others give information about the position of the class through its relationships with the other classes of the model (e.g., number of associations). All these metrics have a strong link with the refactorings presented in the previous section.

2.3. *Heuristic search*

Heuristic search enables to promote discovery or learning [21]. It consists to search a space of possible solutions to a problem or to find an acceptable approximate solution, when an exact algorithmic method is unavailable or too time consuming (e.g., complex combinatorial problems). There are a variety of methods that perform heuristic search as hill climbing [22], simulated annealing [23], GAs [24], and so on. In this section, we give an overview of GAs, and we describe how a GA can be used to generate sequences of refactorings. GA is a powerful heuristic search optimization

DOI: 10.1002/smr

Table I. Considered refactorings in the model refactoring by example approach.

Refactoring name	Description
Extract class	Create a new class and move the relevant fields and methods from the old class into the new class.
Rename method	Rename method with a name that reveals its purpose. This refactoring is intended to give more comprehensiveness to the model design.
Push down method	Move behavior from a superclass to a specific subclass, usually because it makes sense only there.
Push down field	Move a field from super class to a specific subclass, usually because it makes sense only there.
Rename parameter	Rename a parameter within the method parameter list.
Add parameter	Add a new parameter to the method parameter list.
Move field	Move a field from a source class to the class destination when it is more used by the second one than the class on which it is defined.
Move method	Move a method from a class to another one when it is using or used by more features of the destination class than the class on which it is defined.
Pull up method	Move a method from some class(es) to the immediate superclass. This refactoring is intended to help eliminate duplicate methods among sibling classes and hence reduce code duplication in general.
Pull up field	Move a field from some class(es) to the immediate superclass. This refactoring is intended to help eliminate duplicate field declarations in sibling classes.
Extract interface	Create an interface class when many classes use the same subset of a class' interface or two classes have part of their interfaces in common.
Replace inheritance with delegation	Change the inheritance relation by a delegation when the subclass uses only part of a super class' interface or does not want to inherit data.

Table II. Considered metrics in the model refactoring by example approach.

Metric name	Description
Number of attributes (NA)	The total number of attributes of a given class.
Number of private attributes (NPvA)	The total number of private attributes of a given class.
Number of public attributes (NPbA)	The total number of public attributes of a given class.
Number of protected attributes (NProtA)	The total number of protected attributes of a given class.
Number of methods (NMeth)	The total number of methods of a given class.
Number of private methods (NPvMeth)	The total number of private methods in a given class.
Number of public methods (NPbMeth)	The total number of public methods in a given class.
Number of protected methods (NProtMeth)	The total number of protected methods in a given class.
Number of associations (NAss)	The total number of associations.
Number of aggregations (NAgg)	The total number of aggregation relationships.
Number of dependencies (NDep)	The total number of dependency relationships.
Number of generalizations (NGen)	The total number of generalization relationships (each parent–child pair in a generalization relationship).
Number of aggregations hierarchies (NAggH)	The total number of aggregation hierarchies.
Number of generalization hierarchies (NGenH)	The total number of generalization hierarchies.
DIT (DIT)	The DIT value for a class within a generalization hierarchy is the longest path from the class to the root of the hierarchy.
HAgg (HAgg)	The HAgg value for a class within an aggregation hierarchy is the longest path from the class to the leaves.

method inspired by the Darwinian theory of evolution [25]. The basic idea behind GA is to explore the search space by making a population of candidate solutions, also called individuals, evolve toward a

DOI: 10.1002/smr

'good' solution of a specific problem. In GA, a solution can be represented as a vector. Each individual (i.e., a solution) of the population is evaluated by a fitness function that determines a quantitative measure of its ability to solve the target problem. Exploration of the search space is achieved by selecting individuals (in the current population) that have the best fitness values and evolving them by using genetic operators, such as crossover and mutation. The crossover operator insures generation of new children or offspring based on parent individuals. The crossover operator allows transmission of the features of the best fitted parent individuals to new individuals. Each pair of parent individuals produces two children (new solutions). Finally, mutation operator is applied to modify some randomly selected nodes in a single individual. The mutation operator introduces diversity into the population and allows escaping local optima found during the search. Mutation is often performed with a low probability in GAs [24]. Once selection, mutation, and crossover have been applied according to given probabilities, individuals of the newly created generation are evaluated using the fitness function. This process is repeated iteratively, until a stopping criterion is met. This criterion usually corresponds to a fixed number of generations. The result of GA (the best solution found) is the fittest individual produced along all generations.

Hence, to apply GA to a specific problem (i.e., the refactoring problem in our context), the following elements have to be adapted to the problem at hand:

1. representation of the individuals;
2. creation of a population (i.e., a generation) of individuals;
3. evaluation of individuals using a fitness function;
4. selection of the (best) individuals to transmit from one generation to another;
5. creation of new individuals using genetic operators (crossover and mutation) to explore the search space; and
6. generation of a new population using the selected individuals and the newly created individuals.

3. A HEURISTIC SEARCH APPROACH TO MODEL REFACTORING

3.1. Overview of the approach

The approach proposed in this paper exploits examples of model refactorings and a heuristic search technique to automatically suggest sequences of refactorings that can be applied on a given model. The general structure of our approach is illustrated by Figure 1.

Our refactoring approach takes as inputs an initial model and a set of models in the base of examples and their related refactorings and takes as controlling parameters a set of quality metrics. The approach generates a set of refactoring operations that represents refactoring opportunities for the initial model. The process of generating a sequence of refactorings (Figure 2) can be viewed as a mechanism that finds the best way to combine refactoring operations among the list proposed in the models in the base of examples, in such a way to best maximize the similarity between entities to be refactored in the initial model and entities of the models in the base of examples that have undergone the refactoring operations composing the sequence.

Accordingly, the algorithm that generates relevant sequences of refactorings has to explore a huge search space. In fact, the search space is determined by the number of possible refactoring combinations. Formally,

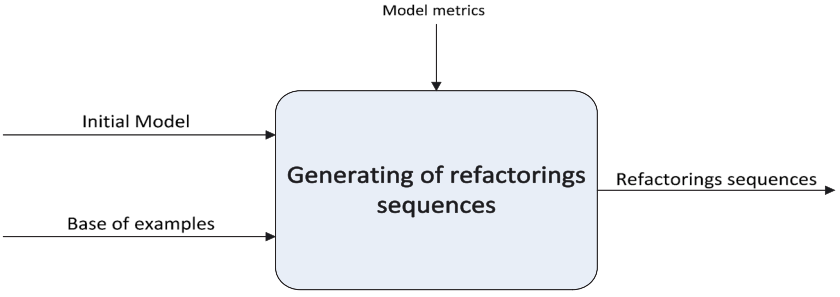

Figure 1. Approach overview.

DOI: 10.1002/smr

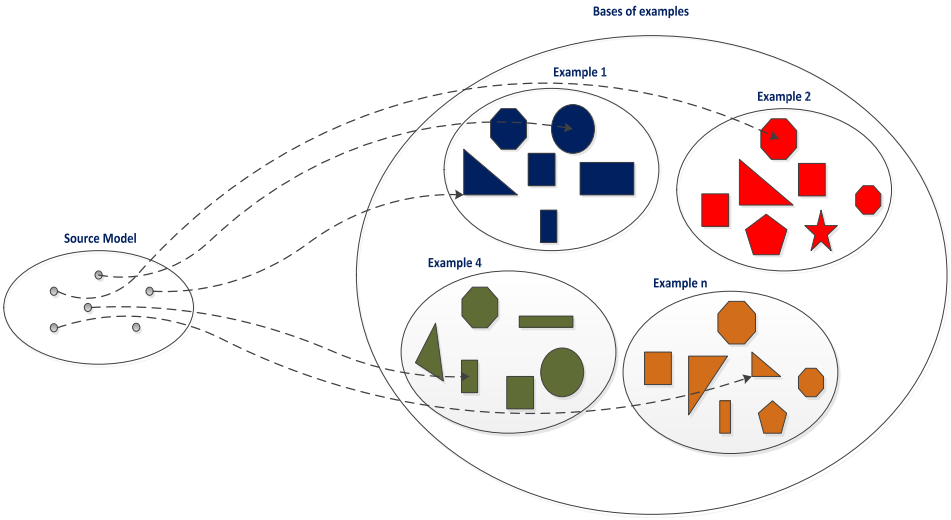

Figure 2. Illustration of proposed generation process.

if m is the number of available refactoring operations, then the number R of possible refactoring subsets is equal to $R = 2^m$. If c is the cardinality of a subset of possible refactorings to which we add the order, then the number of permutations will equal to $c!$. In this context, the number NR of possible combinations that has to be explored by the algorithm is given by

$$NR = \sum_{i=1}^{2^m} c_i!$$

But this brute force method is infeasible in practice, because of the expensive computation. Even for a small number of refactorings (for $m = 5$, NR is 3840), the NR value quickly becomes larger, because the same refactoring operations can be applied several times on different parts of the model (e.g., class, method, and attribute). Because of this large number of possible refactoring solutions, we resorted to a heuristic-based optimization method to solve the problem. Hence, we considered the model refactorings' generation as an optimization problem, and we adapted the GA [25] to this problem in order to find an optimal solution (i.e., a sequence of refactorings) that maximizes the similarity between the entities (class, methods, and attributes) of the initial model and those of the models in the base of examples.

3.2. Adaptation of the genetic algorithm to model refactoring

A high-level view of our adaptation of the GA to the model refactoring problem is given in Figure 3. As this figure shows, the algorithm takes as input a set of quality metrics and a set of model refactoring examples.

Lines 1–3 construct an initial GA population, which is a set of individuals that stand for possible solutions representing sequences of refactorings that can be applied to the classes of the initial model. An individual is a set of triplets; a triplet is called a block, and it contains a class of the initial model denoted as CIM, a class of the base of examples denoted as CBE, and a set of refactorings that were applied to CBE and that are applicable to CIM. To generate an initial population, we start by defining the maximum individual size in terms of a maximum number of blocks composing an individual. This parameter can be specified either by the user or randomly. Thus, the individuals have different sizes. Then, for each individual, the blocks are randomly built; that is, a block is composed by

1. a pair (CIM and CBE) of randomly matched classes, that is, one class(CIM) from the initial model that is under analysis and its randomly matched class (CBE) from the base of examples and

DOI: 10.1002/smr

```
Input: Set of quality metrics
Input: Set of model refactoring examples
Output: A sequence of refactorings
1: I:= set_of(CIM, CBE, set of the refactorings applied to CBE that are applicable to
CIM)
2: P:= set_of(I)
3: initial_population(P, Max_size)
4: repeat
5:    for all I ∈ P do
7:       fitness(I) := Sum [similarity (CIM, CBE) for all (CIM, CBE) belonging to I]
8:    end for
9:       best_solution := best_fitness(I);
10:      P := generate_new_population(P)
11:      it:=it+1;
12: until it=max_it or fitness(best_solution) = 0;
13: return best_solution
```

Figure 3. High-level pseudo-code for genetic algorithm adaptation to our problem.

2. a set of refactorings that we can possibly apply on the class CIM from the initial model extracted from the set of refactorings that were applied to its matched class CBE from the base of examples.

Individuals' representation is explained in more detail in Section 3.3.

Lines 4–13 encode the main GA loop, which explores the search space and constructs new individuals by changing the matched pairs (CIM and CBE) in blocks. During each iteration, we evaluate the quality of each individual in the population. To do so, we use a fitness function that sums the similarities between the classes CMI and CBE of each block composing the individual (line 7). Computation of the fitness function of an individual is described in more detail in Section 3.5. Then, we save the individual having the best fitness (line 9). In line 10, we generate a new population $(p+1)$ of individuals from the current population by selecting 50% of the best fitted individuals from population p and generating the other 50% of the new population by applying the crossover operator to the selected individuals; that is, each pair of selected individuals, called parents, produces two children (new solutions). Then, we apply the mutation operator, with a probability, for both parents and children to ensure the solution diversity; this produces the population for the next generation. The mutation probability specifies how often parts of an individual will mutate. Selection, crossover, and mutation are described in details in Section 3.4.

The algorithm stops when the termination criterion is met (line 12) and returns the best solution found during all iterations (line 13). The termination criteria can be a maximum number of iterations or the best fitness function value. However, the best fitness function value is difficult to predict, and sometimes, it takes a very long time to converge toward this value. Hence, our algorithm is set to stop when it reaches the maximum iteration number or the best fitness function value.

In the following subsections, we describe in details our adaptation of the GA to the model refactoring problem. To illustrate this adaptation, we use an example of a class diagram as a model to refactor. Thus, the base of examples is a set of refactorings' examples on class diagrams.

3.3. Individual representation

An individual is a set of blocks. A block contains three parts as shown by Figure 4: the first part contains the class CIM chosen from the initial model (model under analysis), the second part contains the class CBE from the base of examples that was matched to CIM, and finally the third part contains a list of refactorings that is a subset of the refactorings that were applied to CBE (in its subsequent versions) and that can be applied to CIM.

In our approach, we represented models using predicates. However, we used a slightly different predicate format for representing the classes of the model under analysis and those in the base of examples. Figure 5 illustrates the predicate format used to represent a class (CIM) from the initial model while Figure 6 illustrates the predicate format to represent a class (CBE) from the base of examples. The representation of a CBE class includes a list of refactorings that were applied to this class in a subsequent version of the system's model to which CBE belongs. The subset of a CBE

DOI: 10.1002/smr

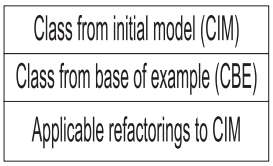

Figure 4. Block representation.

```
Class (Class Name; visibility)
{
        Attribute(Attribute Name; Type; visibility; ReturnType)
        ...
        Method(Method Name; [Parameters]; Visibility; Return Type;)
        ...
        Relation(Name of the source class; Name of the destination class;  Type of the relation)
        ...
}
```

Figure 5. Class representation in the initial model.

```
Class (Class Name; visibility)
{
        Attribute(Attribute Name; Type; visibility; ReturnType)
        ...
        Method(Method Name; [Parameters]; Visibility; Return Type;)
        ...
        Relation(Name of the source class; Name of the destination class;  Type of the relation)
        ...
        Refactoring(Refactoring Name (Parameters))
        ...
}
```

Figure 6. Class representation in the base of examples.

subsequent refactorings that are applicable to a CIM class constitutes the third part of the block having CIM as its first part and CBE as its second part. Hence, the selection of the refactorings to be considered in a block is conformed to some constraints to avoid conflicts and incoherence errors. For example, if we have a *Move_attribute* refactoring operation in the CBE class and the CIM class does not contain any attribute, then this refactoring operation is discarded as we cannot apply it to the CIM class.

The bottom part of Figure 7 shows an example of an individual (i.e., a candidate solution) that we extracted from our experiment described in Section 4. This individual is composed of several blocks. The first block (encircled in Figure 7) was produced by matching a class from the model under analysis (ResourceTreeTable) and a class from the base of example (mxLayoutManager) shown in the top part of Figure 7. Class mxLayoutManager has undergone two refactorings, which can be applied to class ResourceTreeTable. Hence, in this context, the two refactorings are included in the refactoring sequence that constitutes the third part of the first block. It is important to highlight that a class from the initial model can be included only in a single block of a given individual. The top part of Figure 8 shows another example of an individual. Each block of this individual contains one refactoring operation. The bottom part of Figure 8 shows the fragments of an initial model before and after the sequence of refactorings proposed by the individual (at the top of the figure) was applied. Hence the individual represents a sequence of refactoring operations to apply and the classes of the initial model on which they apply.

DOI: 10.1002/smr

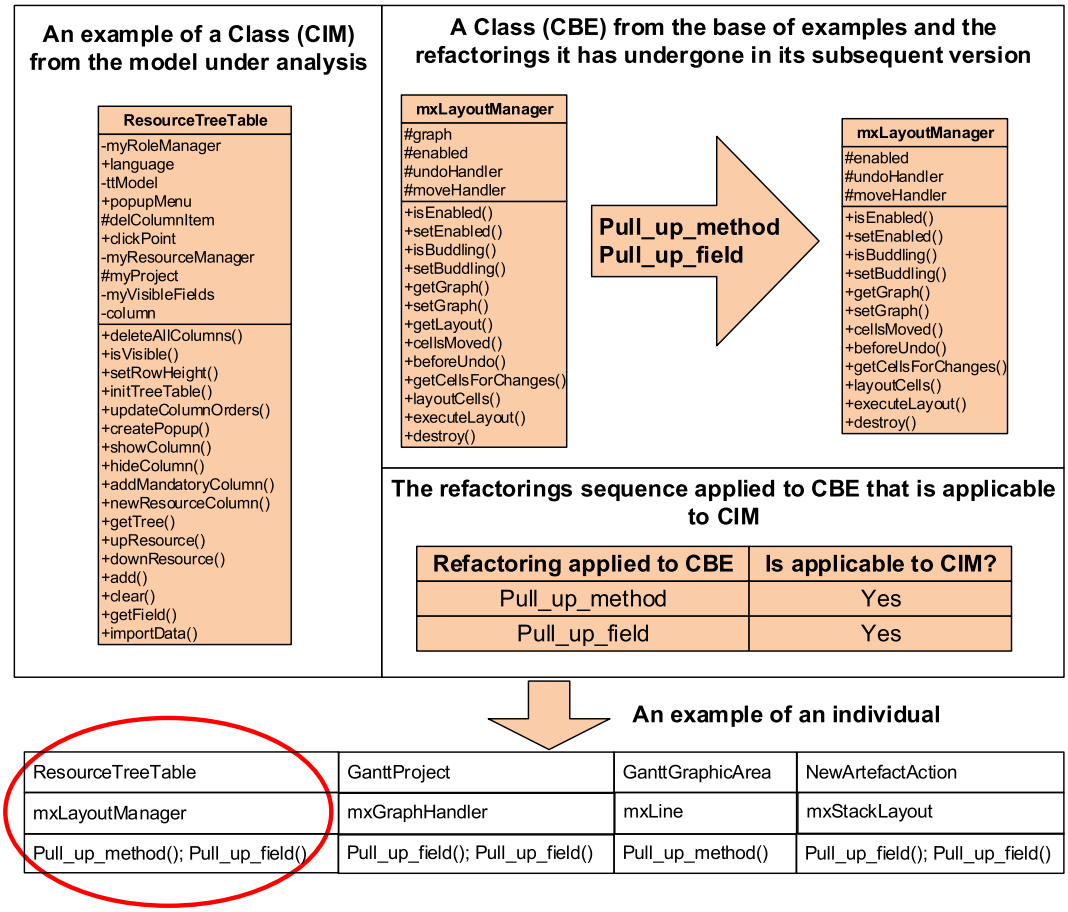

Figure 7. Example extracted from our experiment.

3.4. Genetic operators

3.4.1. Selection. We used the stochastic universal sampling (SUS) [25] to select individuals that will undergo the crossover and mutation operators to produce a new population from the current one. In the SUS, the probability of selecting an individual is directly proportional to its relative fitness in the population. For each iteration, we use SUS to select 50% of individuals from population p for the new population $p+1$. These (population_size/2) selected individuals will be transmitted from the current generation to the new generation and they will 'give birth' to another (population_size/2) new individuals using crossover operator.

3.4.2. Crossover. For each crossover, two individuals are selected by applying the SUS selection [25]. 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'_2. Similarly, the first k refactorings of P_2 become the first k refactorings of P'_1. The rest of the refactorings (from position $k+1$ until the end of the sequence) in each parent P_1 and P_2 are kept. For instance, Figure 9 illustrates the crossover operator applied to two individuals (parents) P_1 and P_2. The position k takes the value 2. The first two refactorings of P_1 become the first two elements of P'_2. Similarly, the first two refactorings of P_2 become the first k refactorings of P'_1.

3.4.3. Mutation. The mutation operator consists of randomly changing one or more dimensions (block) in the solution (vector). Hence, given a selected individual, the mutation operator first

DOI: 10.1002/smr

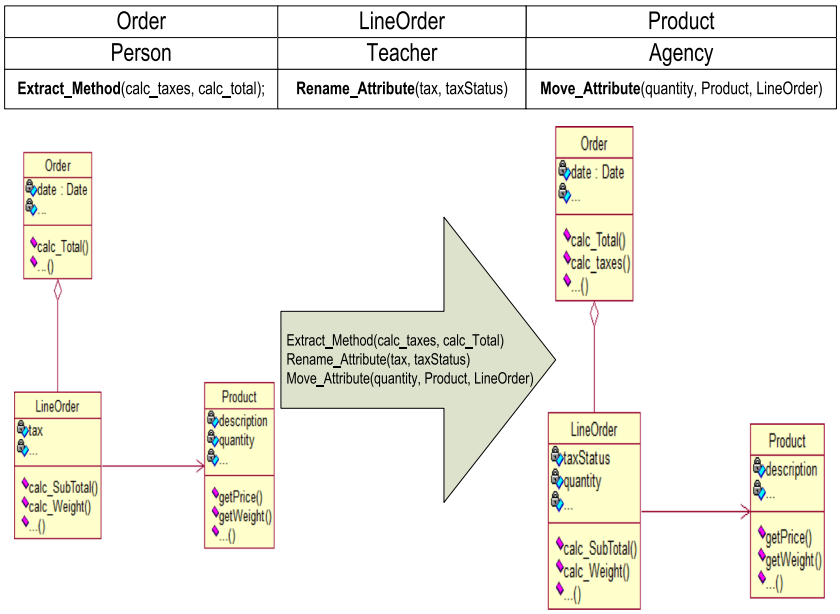

Figure 8. An individual as a sequence of refactorings.

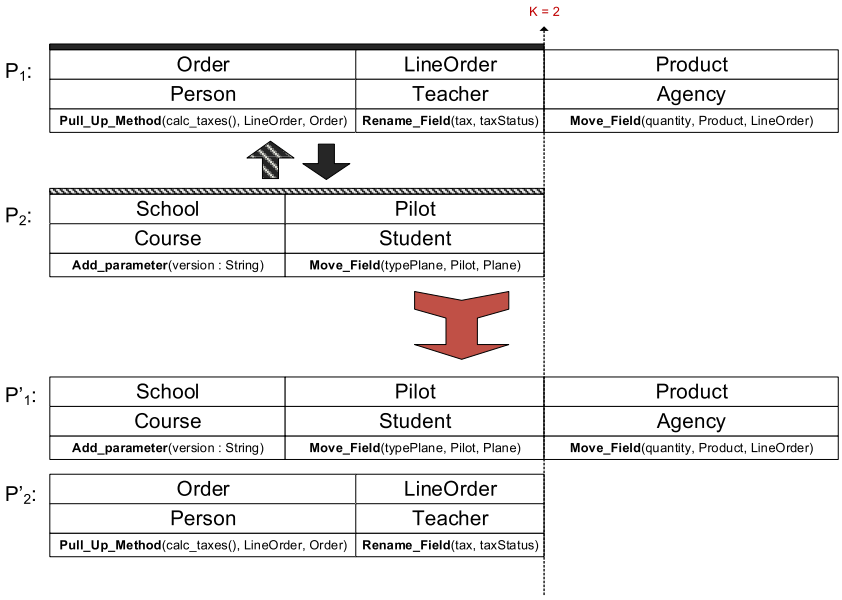

Figure 9. Crossover operator.

randomly selects some blocks in the vector representation of the individual. Then, the CBE of the selected block is replaced by another CBE chosen randomly from the base of examples.

Figure 10 illustrates the effect of a mutation that replaced the refactoring *Rename_Attribute (tax, taxStatus)* applied to the class *LineOrder* (initial model), which is extracted from the class *Teacher* (base of examples) by the refactoring *Rename_Method(calc_SubTotal, calc_TotalLine)* extracted from the new matched class *Student* (base of examples) and applied to the class *LineOrder* (initial model).

DOI: 10.1002/smr

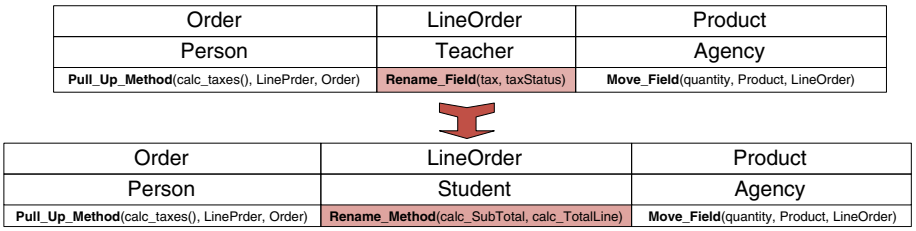

Figure 10. Mutation operator.

3.5. *Decoding of an individual*

The quality of an individual is proportional to the quality of the refactoring operations composing it. In fact, the straight way to evaluate the quality of an individual is to apply its sequence of refactorings to the model under analysis. However, our goal is to find a way to infer correct refactorings using the knowledge that has been accumulated through the refactorings of other models of past projects. Specifically, we want to exploit the similarities between the actual model and other models to infer the sequence of refactorings that we must apply. Our intuition is that a candidate solution that displays a high similarity between the classes of the model and those chosen from the example base should give the best sequence of refactorings.

Practically, the evaluation of an individual should be formalized as a mathematical function called fitness function. The goal is to define an efficient and simple (in the sense not computationally expensive) fitness function in order to reduce the computational complexity. As discussed previously, the fitness function aims to maximize the similarity between the classes of the model in comparison to the ones in the base of examples. In this context, we define the fitness function of a solution as

$$f = \sum_{j=1}^{n} \text{Similarity}\ (CIM,\ CBE) = \sum_{j=1}^{n} \sum_{i=1}^{m} |CIM_i - CBE_i|$$

Where n and m are respectively the number of blocks in the solution and the number of metrics considered in this project. *CIM* and *CBE* are respectively the class from the initial model and the class from the base of examples that belong to the j^{th} block. CIM_i is the i^{th} metric value of the class *CIM* while CBE_i is the i^{th} metric value of the class *CBE*. Figure 11 illustrates the way we compute the similarity between the two given classes using their metrics' values.

To illustrate how the fitness function is computed, we consider a system containing three classes as shown in Table III and a base of examples containing three classes shown in Table IV. In this example, we use six metrics, and these metrics are given for each class in the model in Table III and each class of the base of examples in Table IV.

Consider the example of two individuals I_1 and I_2 respectively composed by one block (*Order* from the model and *Agency* from the BE) and two blocks (*LineOrder*/*Student* and *Product*/*Plane*). The fitness function calculated on these solutions has the value

$$f_{I_1} = |3-4| + |0-4| + \cdots + |1-3| \ = \ 13$$
$$f_{I_2} = (|4-2| + |1-1| + \cdots + |1-0|) + (|2-5| + |2-1| + \cdots + |0-0|) = 12$$

If we consider a population composed by only these two individuals, the evaluation process chooses the one that has the minimum value of fitness function, then I_2 will be chosen as the best individual.

DOI: 10.1002/smr

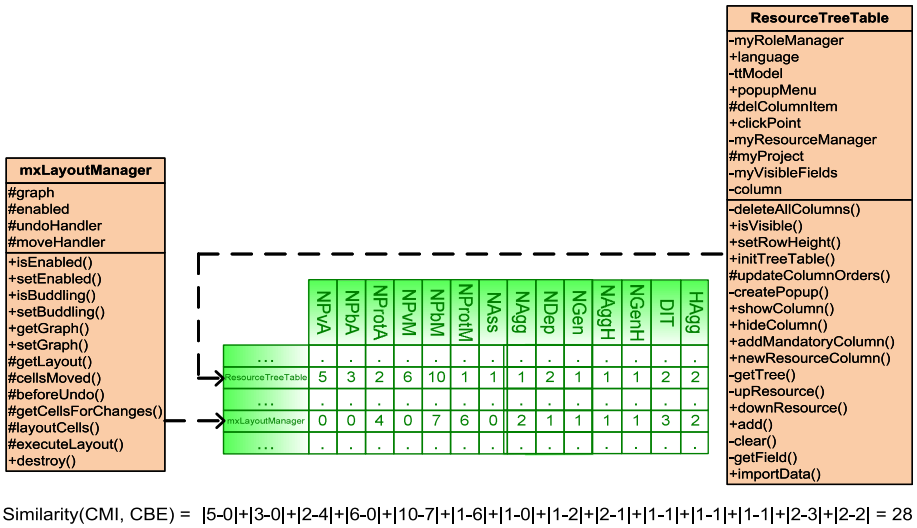

Figure 11. Computing the similarity between two classes.

Table III. Classes from the initial model and their metrics values.

Class in the initial model	NPvA	NPbA	NPbMeth	NPvMeth	NAss	NGen
Order	3	0	5	2	2	1
LineOrder	4	1	3	1	1	1
Product	2	2	6	0	1	0

NPvA, number of private attributes; NPbA, number of public attributes; NPbMeth, number of public methods; NPvMeth, number of private methods; NAss, number of associations; NGen, number of generalizations.

Table IV. Classes from the base of examples and their metrics values.

Class in the base of examples	NPvA	NPbA	NPbMeth	NPvMeth	NAss	NGen
Student	2	1	3	0	3	0
Agency	4	4	1	2	0	3
Plane	5	1	4	0	1	0

NPvA, number of private attributes; NPbA, number of public attributes; NPbMeth, number of public methods; NPvMeth, number of private methods; NAss, number of associations; NGen, number of generalizations.

4. IMPLEMENTATION AND EXPERIMENTAL SETTINGS

In this section, we describe our experimental setup. To set the parameters of GA for the search strategies, we performed several tests, and the final parameters' values were set to a minimum of 1000 iterations for the stopping criterion, to 2 as the minimum length of a solution in terms of number of blocks, and to 25 as the maximum length of a solution. We also set the crossover probability to 0.8 and the mutation probability to 0.5. The crossover probability is 0.9 and the mutation one is 0.5. These values were obtained by trial and error. We selected a high mutation rate because it allows the continuous diversification of the population, which discourages premature convergence to occur.

4.1. Supporting tool

To validate our approach, we implemented a parser that analyzes Java source code and generates a predicate model as illustrated by Figure 5. We used this parser to generate predicate models from

DOI: 10.1002/smr

eight Java open-source projects. To build the base of examples, we completed the generated models by manually entering the refactoring operations extracted with Ref-Finder [26], which these projects have undergone. The Ref-Finder tool allows detection of complex refactorings (68 refactorings) between two program versions using logic-based rules executed by a logic programming engine. Ref-Finder helps finding refactorings that a system has undergone by comparing different versions of the system. We used the refactorings returned by Ref-Finder for two reasons: to build the base of examples and to compute the precision and recall of our approach.

4.2. Research questions

The goal of our experiment is to evaluate the efficiency of our approach in generating relevant sequences of refactorings. In particular, the experiment aimed at answering the following research questions:

RQ1: To what extent can the proposed approach generate the correct sequences of refactorings?
RQ2: Is the approach stable? This question aims to verify if the returned refactorings are correct for different executions of the approach.

To answer RQ1, we evaluated the precision and recall of our approach by applying it on a set of existing projects for which we had several versions and hence information about the refactorings they had undergone. To answer RQ2, we run GA multiple times (31 runs for each project) and observe the algorithm's behavior in terms of precision and recall scores through these executions.

4.3. Selected projects for the analysis

To answer the research questions reported previously, we used eight open-source Java projects to perform our experiments. The projects are the following:

- Ant (v1.8.4): A Java library that is mainly used for building Java applications. Ant provides support to compile, assemble, test, and run Java applications.
- GanttProject (v0.10): A Java project that supports project management and scheduling.
- JabRef (v2.7): A graphical application for managing bibliographical databases.
- JGraphx (v1.10.4.0): A Java Swing diagramming (graph visualization) library.
- JHotDraw (v5.2): A framework for the creation of drawing editors.
- JRDF (v0.5.6.2): A Java library for parsing, storing, and manipulating Resource Description Framework.
- Xerces (v2.5): A set of parsers compatible with XML.
- Xom (v1.2.8): A new XML object model.

We have chosen these open-source projects because they are medium-sized open-source projects and most of them were analyzed in related work (e.g., [14, 26–28]). Most of these open-source projects have been actively developed over the past 10 years. Table V provides some relevant information about these projects.

Table V. Case study settings.

Model	Number of classes	Number of methods	Number of attributes	Number of expected refactoring operations
Ant 1.8.4	824	2090	1048	139
GanttProject 2.0.10	479	960	495	91
JabRef 2.7	594	253	237	32
JGraphx 1.10.4.0	191	1284	420	96
JHotDraw 5.2	160	519	141	71
JRDF v0.5.6.2	734	19	10	41
Xerces 2.5	625	2113	1408	182
Xom 1.2.8	252	186	31	36

DOI: 10.1002/smr

In our validation, we use one project as the system under analysis and the other seven projects as the base of examples. Then, we compare the refactoring sequences returned by the algorithm with the ones returned by Ref-Finder when executed on the same version of the system under analysis and the following version.

4.4. *Measures of precision and recall*

To assess the accuracy of our approach, we compute the measures precision and recall originally stemming from the area of information retrieval. When applying precision and recall in the context of our study, the precision denotes the fraction of correctly detected refactoring operations among the set of all detected operations. The recall indicates the fraction of correctly detected refactoring operations among the set of all actually applied operations (i.e., how many operations have not been missed). In general, the precision denotes the correctness of the approach (i.e., the probability that a detected operation is correct), and the recall denotes the completeness of the approach (i.e., the probability that an actually applied operation is detected). Both values may range from 0 to 1, whereas a higher value is better than a lower one.

5. RESULTS AND DISCUSSION

In this section, we present the results of our experiment. We specifically discuss the results of our GA algorithm in terms of precision and recall and in terms of its stability. We also assess the effectiveness of our approach by comparing it with two other approaches. Finally, we discuss some threats to the validity of the results of our experiment.

5.1. *Precision and recall*

The precision and recall results might vary depending on the refactorings used, which are randomly generated, though guided by a meta-heuristic. We chose two projects (Xerces 2.5 and JHotDraw 5.2) to illustrate the results given by our approach. Figures 12 and 13 show the results of multiple executions (31 executions) of our approach on Xerces and JHotDraw, respectively. Each of these figures displays the precision and the recall values for each execution.

Generally, the average precision and recall for all projects (around 85%) allows us to positively answer our first research question RQ1 and conclude that the results obtained by our approach are very encouraging. The precision, which is sometimes close to 100%, proves that all the refactorings proposed by our approach were indeed applied to the system's model in its subsequent version (i.e., the proposed refactorings match those returned by Ref-Finder when applied on the system's model and its subsequent version).

Despite the good results, we noticed a very slight decrease in recall versus precision in some projects; this is illustrated by Figure 12 for the Xerces project. We made a further analysis of the results to find out the factors behind this decline. Our analysis pointed out toward two important

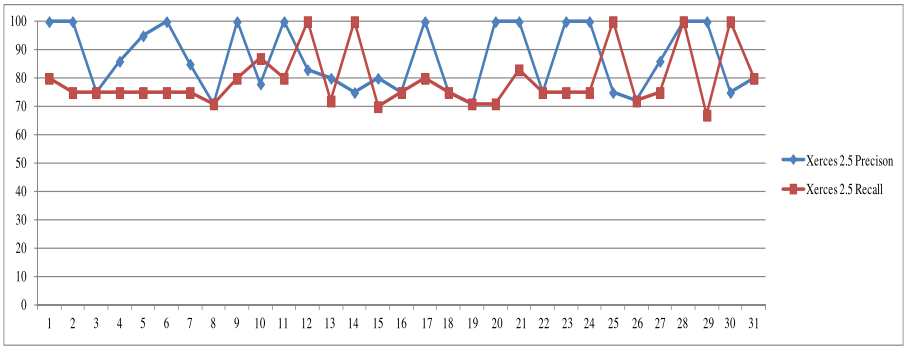

Figure 12. Multiple execution results for Xerces project.

DOI: 10.1002/smr

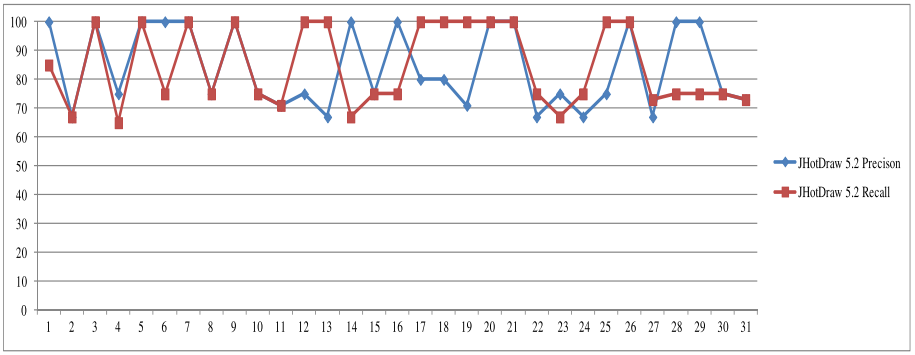

Figure 13. Multiple execution results for JHotDraw project.

factors. The first factor is the project domain. In this study, we tried to propose refactorings using a base of examples that contains different projects from different domains. We noticed that some projects focus on some types of refactorings compared with others (i.e., some projects in the base of examples have a big frequency of «*pull_up_field*» and «*pull_up_method*»). The second factor is the number and types of refactorings (i.e., 12) considered in this experimentation. Indeed, we noticed that the refactorings («*pull_up_method*», «*pull_up_field*», «*add_parameter*», «*extract_class*», «*push_down_field*», «*push_down_method*», «*rename_parameter*», «*rename_method*», and «*move field*») are located correctly in our approach. We have no certainty that these factors can improve the results, but we consider analyzing them as a future work to further clarify many issues.

5.2. *Stability*

To ensure that our results are relatively stable, we compared the results of multiple executions of the approach on each of the eight open-source projects. Figure 14 shows the precision results of these multiple executions for all the projects while Figure 15 shows an error bar plot displaying the minimum precision, the maximum precision , and the average precision of these executions for each project. Similarly, Figure 16 shows the recall results of the 31 executions for all the projects while Figure 17 shows an error bar plot displaying the minimum recall, the maximum recall, and the average recall of these executions for each project. The intervals displayed by Figures 15 and 17 confirm that precision and recall scores are approximately the same for different executions in all the projects in the base of examples. The range between the minimum and the maximum values for each project is not large. For example, for JGraph and Ant projects, the range is around 20%, and it is stable at around 30% for the rest of the projects. In Figure 15, 75% of the projects have an average of precision within the range 80–90%. In Figure 17, seven projects among eight have an average within the range 80–90%. This analysis allows us to conclude that our approach is stable, which positively answers our second research question RQ2.

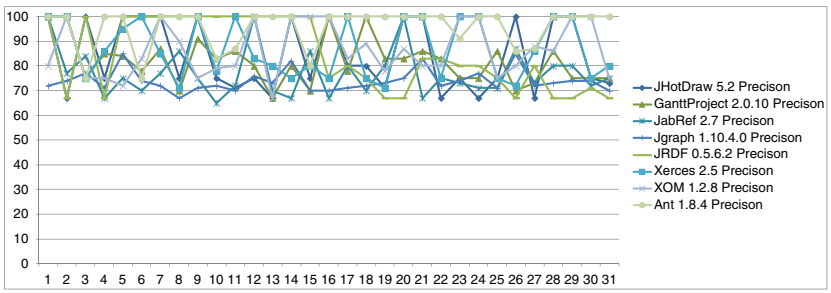

Figure 14. Multiple execution precision results of eight open-source projects in our approach.

DOI: 10.1002/smr

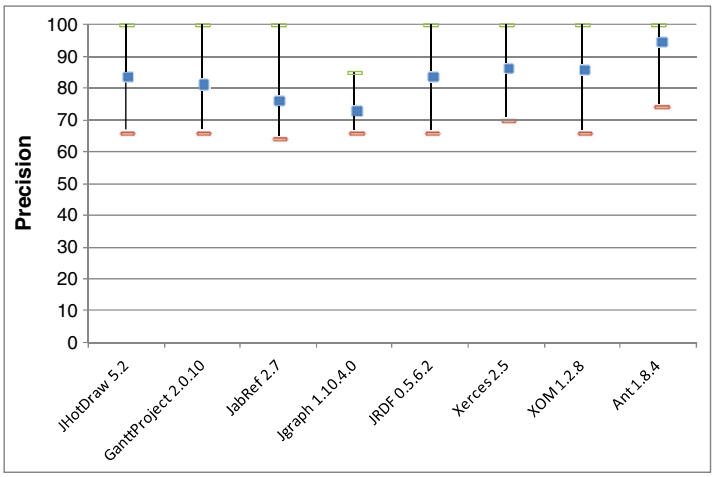

Figure 15. Error bar chart for the precision.

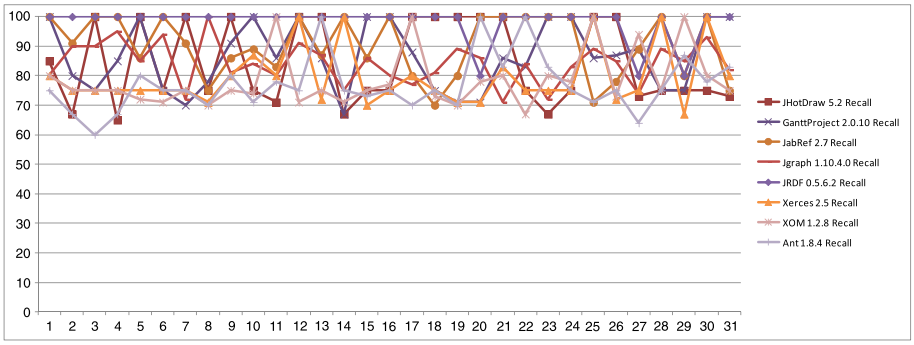

Figure 16. Multiple execution recall results of eight open-source projects in our approach.

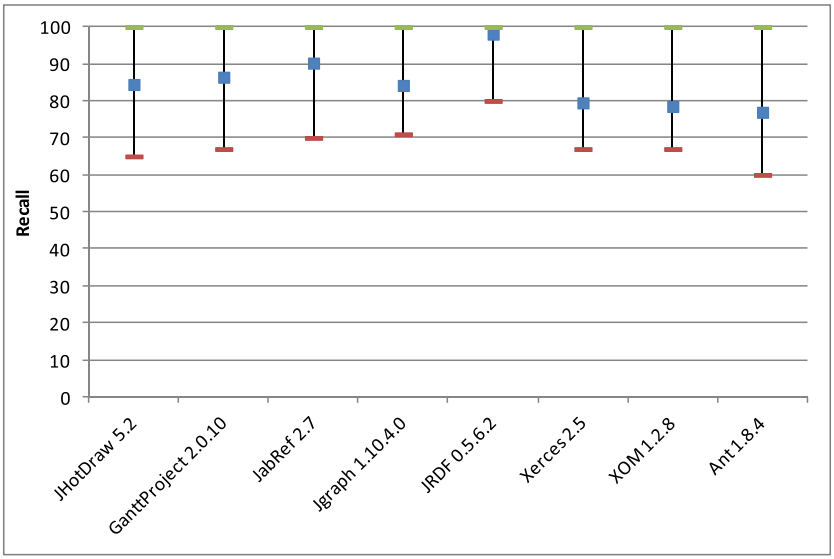

Figure 17. Error bar chart for the recall.

DOI: 10.1002/smr

5.3. *Effectiveness of our approach*

To assess the effectiveness of our approach, we conducted a comparative study between our approach and two other approaches: (1) a random search approach and (2) the algorithm proposed by Kessentini *et al*. in [29]. For the purpose of the first comparison, we implemented an algorithm that randomly selects pairs of CIM/CBEs. We run the random search algorithm under the same conditions in which we performed the experiment with our approach. Figures 18 and 19 illustrate the results of multiple executions (31 executions) of the random search algorithm on the same eight projects we used in our experiment. While the average precision and recall of our approach is around 85%, both precision and recall values of the 31 executions of the random search algorithm do not exceed 50%; that is, these values vary between 20% and 50%. We consequently conclude that our approach is more effective than an equivalent random search approach.

We also compared our approach with the approach proposed in [29] where genetic programming (GP) is used to generate detection rules based on quality metrics. In fact, because the used algorithms are meta-heuristics, thus they can produce different results on every run when applied to the same problem instance. To this end, the use of rigorous statistical tests is essential to provide support to the conclusions derived by analyzing such data [30]. For this reason, we independently performed 31 executions using the algorithm in [29] for each of the eight open-source projects that we used in our experiment, and we used the *p*-values of the Wilcoxon rank-sum test [31] as a

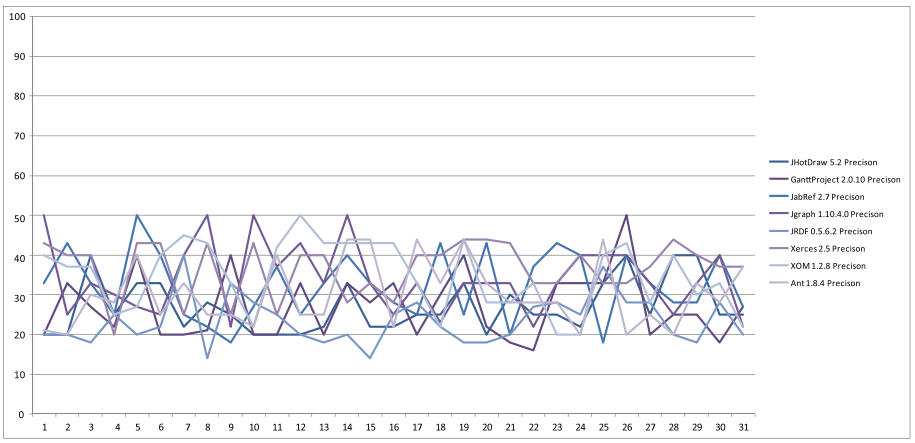

Figure 18. Precision of multiple executions of the random search algorithm.

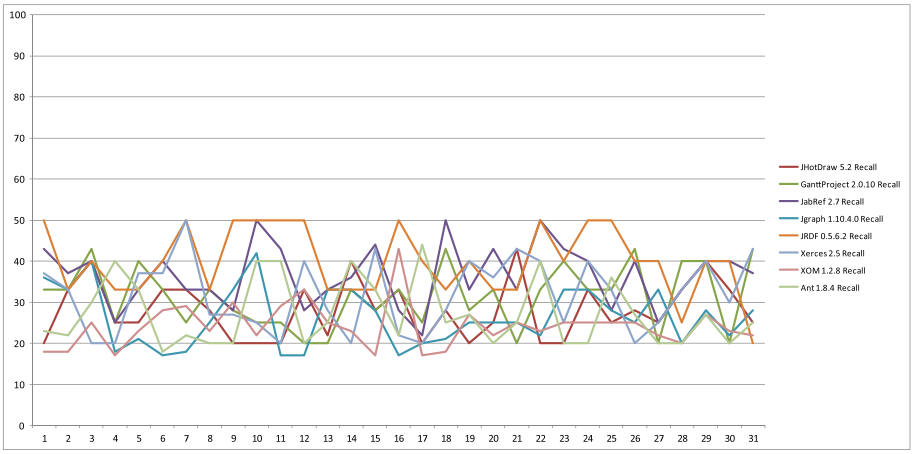

Figure 19. Recall of multiple executions of the random search algorithm.

DOI: 10.1002/smr

statistical test to compare the results of the two algorithms. In our context, a p-value that is less than or equal to α (=0.05) means that the distributions of the results of the two algorithms are different in a statistically significant way. In fact, we computed the p-value of GP obtained results compared with our approach. In this way, we could decide whether the outperformance of our approach over the GP approach is statistically significant. Table VI displays the precision and recall median values of our algorithm (MOREX) and the GP algorithm for the eight open-source projects. The p-value for the precision median results of GP compared with our approach is 0.0188 while the p-value of the recall median results of GP compared with our approach is 0.0181. Consequently, as these values are less than α (= 0.05), we conclude that the precision and recall median values of our algorithm are statistically different from the GP ones on each of the systems. As Table VI shows, it is clear that MOREX outperforms the approach in [29] over all the open-source systems.

5.4. Threats to validity

We have some points that we consider as threats to the generalization of our approach. The most important one is the use of the Ref-Finder tool to build the base of examples, and at the same time, we compare the results obtained by our algorithm with those given by Ref-Finder. Another factor that could have been of influence on the obtained results is the sets of metrics and refactorings that we considered in our experiment. We made a preliminary analysis to select refactorings that apply at the model level, and we accordingly choose a set of related metrics. However, further analysis is needed to build a catalog of refactorings that are applicable in designing models and in identifying metrics that are impacted by these refactorings.

An important consideration is the impact of the example base size on the quality of refactoring solutions. In general, our approach does not need a large number of examples to obtain good detection results. The reliability of the proposed approach requires an example set of applied refactoring on different systems. It can be argued that constituting such a set might require more work than these examples. In our study, we showed that by using some open-source projects, the approach can be used out of the box and will produce good refactoring results for the studied systems. However, we agree that, sometimes, within specific contexts, it is difficult to define and find opportunities of refactorings. In an industrial setting, we could expect a company to start with some few open-source projects and gradually migrate its set of refactoring examples to include context-specific data. This might be essential if we consider that different languages and software infrastructures have different best/worst practices.

Finally, because we viewed the model refactorings' generation problem as a combinatorial problem addressed with heuristic search, it is important to contrast the results with the execution time. We executed our algorithm on a standard desktop computer (i7 CPU running at 2.67 GHz with 8 GB of RAM). The execution time for refactorings' generation with a number of iterations (stopping criteria) fixed to 1000 was less than 3 min. This indicates that our approach is reasonably scalable from the performance standpoint. However, the execution time depends on the number of refactorings and the size of the models in the base of examples.

Table VI. Precision and recall median values of genetic programming [29] and model refactoring by example over 31 independent simulation runs.

Models	Precision MOREX (%)	Precision GP (%)	Recall MOREX (%)	Recall GP (%)
Ant 1.8.4	78	72	81	77
GanttProject 2.0.10	82	78	84	82
JabRef 2.7	84	82	79	71
JGraphx 1.10.4.0	87	82	84	82
JHotDraw 5.2	86	81	86	81
JRDF v0.5.6.2	81	79	81	77
Xerces 2.5	82	77	83	81
Xom 1.2.8	86	79	87	78

MOREX, model refactoring by example; GP, genetic programming.

DOI: 10.1002/smr

6. RELATED WORK

Much work has been done on source code refactoring. The best way to correct the source code is to analyze it and to propose the appropriate refactorings to correct the defects it may contain [18]. This method is very expensive in terms of time and resources. Consequently, many approaches were proposed to (semi)automatically support source code refactoring (e.g., [8, 14, 32, 33]). These approaches use different techniques and strategies. For example, the work in [8] analyzed the best-case and worst-case impacts of refactorings on coupling and cohesion dimensions. Most of the considered refactorings are applied at the code source level (e.g., Move Method, Replace Method with Method Object, Replace Data Value with Object, and Extract Class). The approach in [14] proposed to represent code smells and use these representations to generate appropriate refactoring rules that can be automatically applied to source code. In [33], program invariants are used to detect a specific point in the program to apply refactoring, and an invariant pattern matcher was developed and used on an existing Java code base to suggest some common refactorings.

Model refactoring is still at a relatively young stage of development. Most of the existing approaches for automating refactoring activities at the model level are based on rules that can be expressed as assertions (i.e., invariants, pre-condition and post-condition) [34, 35] or graph transformations targeting refactoring operations in general (e.g., [36, 37]) or refactorings related to design patterns' applications (e.g., [9]). The use of invariants [34] has been proposed to detect some parts of the model that require refactoring. Refactorings are expressed using declarative rules. However, a complete specification of refactorings requires an important number of rules, and the refactoring rules must be complete, consistent, non-redundant, and correct. In [9], refactoring rules are used to specify design patterns' applications. In this context, design problems solved by these patterns are represented using models, and the refactoring rules transform these models according to the solutions proposed by the patterns. However, not all design problems are representable using models; that is, for some patterns, the problem space is quite large, and the problem cannot be captured in a single or a handful of problem models [9]. Finally, an issue that is common to most of these approaches is the problem of sequencing and composing refactoring rules. This is related to the control of rules' applications within rule-based transformational approaches in general.

Our approach is inspired by contributions in search-based software engineering (SBSE) (e.g. [38–42]). As the name indicates, SBSE uses a search-based approach to solve optimization problems in software engineering. Techniques based on SBSE are a good alternative to tackle many of the aforementioned issues [40]. For example, a heuristic-based approach is presented in [38, 39] in which various software measures are used as indicators for the need of a certain refactoring. In [41], a GA is used to suggest refactorings to improve the class structure of a system. The algorithm uses a fitness function that relies on a set of existing object-oriented metrics. Harman and Tratt [39] propose to use the Pareto optimality concept to improve search-based refactoring approaches when the evaluation function is based on a weighted sum of metrics. Both the approaches in [41] and [39] were limited to the Move Method refactoring operation. In [38], the authors present a comparative study of four heuristic search techniques applied to the refactoring problem. The fitness function used in this study was based on a set of 11 metrics. The results of the experiments on five open-source systems showed that hill climbing performs better than the other algorithms. In [42], the authors proposed an automated refactoring approach that uses GP to support the composition of refactorings that introduce design patterns. The fitness function used to evaluate the applied refactorings relies on the same set of metrics as in [38] and a bonus value given for the presence of design patterns in the refactored design. Our approach can be seen as linked to this approach as we aim at proposing a combination of refactorings that must be applied to a design model. Our work is more related to the work in [40] where the authors proposed a by-example approach based on search-based techniques for model transformation. A particle swarm optimization algorithm is used to find the best subset of transformation fragments in the base of examples that can be used to transform a source model (i.e., Class Diagram) to a target model (i.e., Relational Schema). Hence, this approach targets exogenous transformations (i.e., different source and target languages) while our proposal MOREX is dedicated to refactorings that are endogenous transformations that aim at correcting design defects. Furthermore, the fitness function proposed in [40] relies on the adequate mapping of the selected transformation examples with the constructs of the

DOI: 10.1002/smr

model (e.g., class and relationship) to be transformed while our fitness function exploits the structural similarity between classes. To conclude, in our contribution, we propose to use a different meta-heuristic algorithm to a different problem than the one in [40] with a new adaptation (fitness function, change operators, etc.).

7. CONCLUSION AND FUTURE WORK

In this paper, we introduced MOREX, an approach to automate model refactoring using heuristic-based search. The approach considers refactoring as an optimization problem, and it uses a set of refactoring examples to propose appropriate sequences of refactorings that can be applied on a source model. MOREX randomly generates sequences of applicable refactorings and evaluates their quality depending on the similarity between the source model and the examples of models at hand.

We have evaluated our approach on real-world models extracted from eight open-source systems. The experimental results indicate that the proposed refactorings are comparable with those expected; that is, the proposed refactorings match those returned by the Ref-Finder tool when applied on a model and its subsequent version. We also performed multiple executions of the approach on the eight open-source projects, and the results have shown that the approach is stable regarding its precision and recall.

While the results of the approach are very promising, we plan to extend it in different ways. One issue that we want to address as a future work is related to the base of examples. In the future, we want to extend our base of examples to include more refactoring operations. We also want to study and analyze the impact of using domain-specific examples on the quality of the proposed sequences of refactorings. Actually, we kept the random aspect that characterizes GAs even in the choice of the projects used in the base of examples without prioritizing one or more specific projects on others to correct the one under analysis.

We also plan to compare our results with other existing approaches other than the Ref-Finder tool and perform a further analysis on the nature and type of refactorings that are easier or harder to detect. In addition, the evaluation of the sequences of refactorings returned by our approach was based on the similarity between the classes of the source model and the classes from the base of examples. However, only the syntactic aspect was considered when computing these similarities; that is, the similarity was based on a set of metrics that are mostly related to the structural features of the classes (e.g., number of attributes, number of methods, etc.). In the future, we plan to study the semantic properties (e.g., similarity of classes' names) that can be used as similarity or dissimilarity factors to enhance our evaluation function.

REFERENCES

1. Seacord RC, Plakosh D, Lewis GA. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*, Addison-Wesley Longman Publishing Co., Inc: Boston, MA, USA, 2003; 368.
2. Lientz BP, Swanson EB, Tompkins GE. Characteristics of application software maintenance. *Communications of the ACM* 1978; **21**(6):466–471.
3. ISO/IEC. International Standard - ISO/IEC 14764 IEEE Std 14764-2006 Software Engineering; Software Life Cycle Processes &; Maintenance. ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998), 2006: p. 0_1-46.
4. Fowler M. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley: Boston, MA, USA, 1999.
5. Mens T, Tourwé T. A survey of software refactoring. *IEEE Transactions on Software Engineering* 2004; **30**(2):126–139.
6. Opdyke WF. *Refactoring: A Program Restructuring Aid in Designing Object-oriented Application Frameworks*, University of Illinois at Urbana-Champaign, 1992.
7. Moha N. *DECOR: Détection et correction des défauts dans les systèmes orientés objet*, Université de Montréal & Université des Sciences et Technologies de Lille: Montréal, 2008; 157.
8. Bois BD, Demeyer S, Verelst J. Refactoring–improving coupling and cohesion of existing code. *Proceedings of the 11th Working Conference on Reverse Engineering*, IEEE Computer Society, 2004; 144–151.
9. El Boussaidi G, Mili H. Understanding Design Patterns—What is the Problem? *Software: Practice and Experience* 2012; **42**(12):1495–1529.
10. Mens T, Taentzer G, Muller D. Challenges in model refactoring. *Proc. 1st Workshop on Refactoring Tools*, University of Berlin, 2007.

DOI: 10.1002/smr

11. Zhang J, Lin Y, Gray J. Generic and Domain-specific Model Refactoring Using a Model Transformation Engine. *Model-driven Software Development – Research and Practice in Software Engineering*. Springer: Hiedeberg, 2005.
12. Douglas CS. Model-driven engineering. *IEEE Computer* 2006; **39**(2):41–47.
13. El-Boussaidi G, Mili H. Detecting patterns of poor design solutions using constraint propagation. *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, Springer-Verlag: Toulouse, France, 2008; 189–203.
14. Moha N, *et al*. DECOR: a method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions* 2008; **36**(1):20–36.
15. Munro MJ. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *Software Metrics, 2005. 11th IEEE International Symposium*, 2005.
16. Marinescu R. Detection strategies: metrics-based rules for detecting design flaws. *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference*, 2004.
17. Kessentini M, *et al*. Search-based design defects detection by example. *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software*, Springer-Verlag: Saarbrücken, Germany, 2011; 401–415.
18. Fowler M, Beck K. Refactoring: improving the design of existing code. *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods*, Springer-Verlag, 1999; 256.
19. Fenton NE, Pfleeger ASL. *Software Metrics: A Rigorous and Practical Approach* (2nd edn) PWS Publishing Co.: Boston, MA, USA ©, 1998; 656.
20. Genero M, Piattini M, Calero C. *Empirical Validation of Class Diagram Metrics*. *Empirical Software Engineering, Proceedings. International Symposium*, 2002.
21. Pearl J. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley Longman Publishing Co., Inc: Boston, MA, USA, 1984; 382.
22. Mitchell M. *An Introduction to Genetic Algorithms*, MIT Press: Cambridge, MA, USA, 1998; 209.
23. Kirkpatrick S, Gelatt CD, Vecchi MP. Optimization by simulated annealing. *Science* 1983; **220**:671–680.
24. Goldberg DE. *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Longman Publishing Co., Inc.: 1989; 372.
25. Koza JR. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press: Cambridge, MA, USA, 1992; 680.
26. Kim M, *et al*. Ref-Finder: a refactoring reconstruction tool based on logic query templates. *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM: Santa Fe, New Mexico, USA, 2010; 371–372.
27. Ghannem A, Kessentini M, El-Boussaidi G. Detecting Model Refactoring Opportunities Using Heuristic Search. *in Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '11)*, Litoiu M, Stroulia E and MacKay S, (eds). IBM Corp.: Riverton, NJ, USA, 2011; 175–187.
28. Ouni A, *et al*. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering* 2013; **20**(1):47–79.
29. Kessentini M, *et al*. Design defects detection and correction by example. *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, IEEE Computer Society, 2011; 81–90.
30. Arcuri A, Briand L. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 2012; DOI:10.1002/stvr.1486.
31. Wilcoxon F. Individual comparisons by ranking methods. *Breakthroughs in Statistics*, Kotz S, Johnson N (eds). Springer: New York, 1992; 196–202.
32. Hui L, *et al*. Facilitating software refactoring with appropriate resolution order of bad smells, in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, 2009: Amsterdam, The Netherlands, p. 265–268.
33. Kataoka Y, *et al*. Automated support for program refactoring using invariants. *Software Maintenance, Proceedings. IEEE International Conference*, 2001.
34. Van Der Straeten R, Jonckers V, Mens T. A formal approach to model refactoring and model refinement. *Software and Systems Modeling* 2007; **6**(2):139–162.
35. Van Kempen M, *et al*. Towards proving preservation of behaviour of refactoring of UML models, in *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries (SAICSIT '05)*, 2005: South African Institute for Computer Scientists and Information Technologists, Republic of South Africa. p. 252–259.
36. Mens T, Taentzer G, Runge O. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling* 2007; **6**(3):269–285.
37. Biermann E. EMF model transformation based on graph transformation: formal foundation and tool environment. *Proceedings of the 5th International Conference on Graph Transformations*, Springer-Verlag: Enschede, The Netherlands, 2010; 381–383.
38. O'Keeffe M. Search-based refactoring: an empirical study. *Journal of Software Maintenance and Evolution* 2008; **20**(5):345–364.
39. Harman M, Tratt L. Pareto optimal search based refactoring at the design level. *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, ACM: London, England, 2007; 1106–1113.
40. Kessentini M, *et al*. Search-based model transformation by example. *Software & Systems Modeling* 2012; **11**(2):209–226.

DOI: 10.1002/smr

41. Seng O, Stammel J, Burkhart D. Search-based determination of refactorings for improving the class structure of object-oriented systems. *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, ACM: Seattle, Washington, USA, 2006; 1909–1916.
42. Jensen AC, Cheng BHC. On the use of genetic programming for automated refactoring and the introduction of design patterns. *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, ACM: Portland, Oregon, USA, 2010; 1341–1348.

AUTHORS' BIOGRAPHIES

Adnane Ghannem is a PhD student at ETS-Canada. He is working on the application of artificial intelligence techniques in software engineering. His research interests include model-driven engineering, model refactoring, and software quality.

Ghizlane El Boussaidi is an associate professor in software engineering at the department of software and IT engineering of ETS-Canada. Her research interests include software architecture and design, software re-engineering and modernization, model-driven development, and model refactoring and transformation.

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 PhD in Computer Science, University of Montreal (Canada), 2011. His research interests include the application of artificial intelligence techniques to software engineering (search-based software engineering), software testing, model-driven engineering, software quality, and re-engineering. He has published around 50 papers in conferences, workshops, books, and journals including three best paper awards. He has served as program committee member in several major conferences (GECCO, ICMT, CLOSER, CAL, LMO, MODELS, etc.) and as organization member of many conferences and workshops (LMO, MDEBE, etc.). He is also the co-chair of SBSE track at the prestigious conference GECCO2014 (22nd International Conference on Genetic Algorithms and the 18th Annual Genetic Programming Conference).

DOI: 10.1002/smr