Recommending Model Refactoring Rules from Refactoring Examples

Chihab eddine Mokaddem
Université de Montréal
Montréal, QC, Canada
mokaddec@iro.umontreal.ca

Houari Sahraoui
Université de Montréal
Montréal, QC, Canada
sahraouh@iro.umontreal.ca

Eugene Syriani
Université de Montréal
Montréal, QC, Canada
syriani@iro.umontreal.ca

ABSTRACT

Models, like other first-class artifacts such as source code, are maintained and may be refactored to improve their quality and, consequently, one of the derived artifacts. Considering the size of the manipulated models, automatic support is necessary for refactoring tasks. When the refactoring rules are known, such a support is simply the implementation of these rules in editors. However, for less popular and proprietary modeling languages, refactoring rules are generally difficult to define. Nevertheless, their knowledge is often embedded in practical examples. In this paper, we propose an approach to recommend refactoring rules that we lean automatically from refactoring examples. The evaluation of our approach on three modeling languages shows that, in general, the learned rules are accurate.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; **Search-based software engineering**;

KEYWORDS

model refactoring, genetic programming, by-example approach, search-based software engineering

ACM Reference Format:
Chihab eddine Mokaddem, Houari Sahraoui, and Eugene Syriani. 2018. Recommending Model Refactoring Rules from Refactoring Examples. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239406

1 INTRODUCTION

Model-driven engineering is increasingly popular in industry [23]. In industrial contexts, the complexity of models keeps on increasing and they are used in various development and maintenance activities. Like other first-class artifacts such as source code, they are maintained and refactored to improve their quality. Considering the size of the manipulated models, automatic support is necessary to refactor these models.

Much work has been done on model refactoring [24]. Most of the contributions can be classified into two families: (1) tools and mechanisms to define and apply refactorings [17, 22, 25, 29, 31, 37], and (2) specific refactoring rule definition [6, 11, 34]. When the languages (metamodels), in which models are expressed, are of general purpose such as UML, there is a critical mass of researchers and users that allow to shape, test, and prove refactoring rules. This is not the case for domain-specific languages (DSLs), where it is not always possible to have such a critical mass to define refactoring rules, due to the specific expertise needed. Thus knowledge is not available to feed the refactoring tools.

When fully writing refactoring rules for DSLs is difficult, if not impossible, a promising alternative is to learn them from examples. This idea was successfully investigated for learning model transformation rules from examples [1, 9]. As model refactoring can be seen as a particular use of model transformation [18, 19], one can adapt these learning algorithms for refactoring. However, this adaption is difficult for two reasons. First, refactoring is an inplace model transformation. Most of the existing by example techniques are intended to generate a completely new model. Second, not the entire source model is affected by the transformation. Only specific situations in the model constitute refactoring opportunities.

In this paper, we propose an approach to recommend refactoring rules learned from examples. Our approach can apply to different scenarios. For example, a modeler starts performing refactoring on a large model. Then, after some occurrences, she feeds in the changed model fragments (before and after the changes) into our approach. The learning process can then suggest rules that she can apply (with or without modifications) to the rest of the model. Another scenario is to collect different versions of models on which manual refactoring had been applied in the past. Then, starting from these model versions, our approach derive refactoring rules to potentially apply on new models. In these scenarios, our approach does not pretend to provide absolute correct refactoring rules. Instead, it suggests the rules that best conform to the provided examples.

We view rule learning as an optimization problem and we solve it using genetic programming. Our algorithm searches for refactoring rules that best conform to the provided examples. Examples are pairs of models (or model fragments)

MODELS '18, October 14–19, 2018, Copenhagen, Denmark
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-4949-9/18/10...$15.00
https://doi.org/10.1145/3239372.3239406

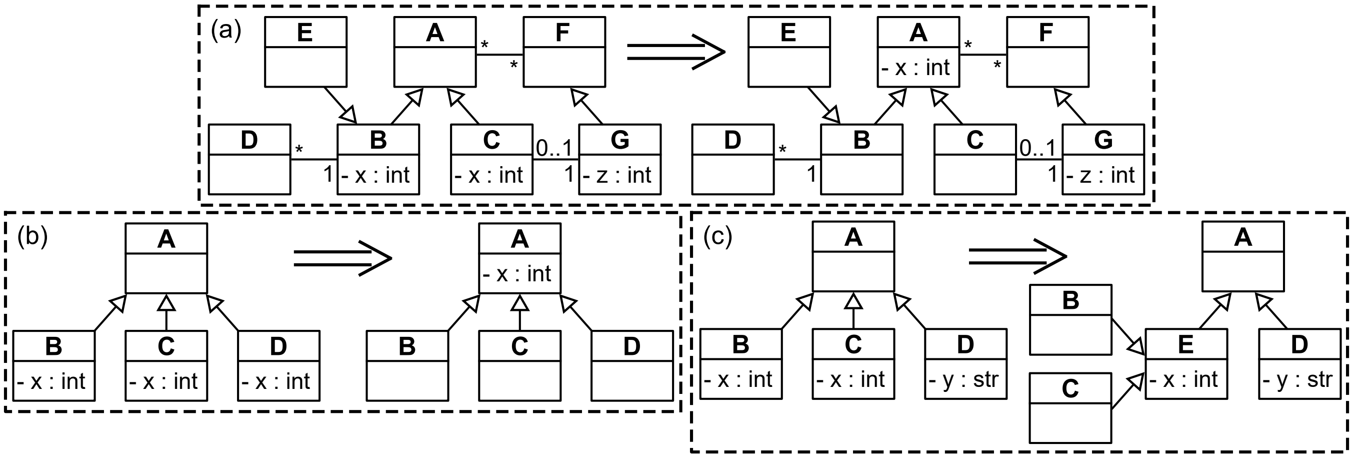

Figure 1: Three examples to learn rules for the Pull-up field refactoring

before and after the refactoring. Our assumption is that it is easier for domain experts to provide concrete examples of situations where a refactoring must be applied and how to apply it, than defining fully-fledged, consistent, and general refactoring rules.

To evaluate our approach, we applied it to three known metamodels for which we have the sought refactoring rules beforehand (ground truth).These metamodels have different kinds of refactoring with different complexities. Our results show that it is possible to learn complex refactoring rules, but the accuracy of these rules depends on the coverage of the provided examples.

In Section 2, we explain the challenges to learn refactoring transformations from examples. In Section 3, we present our approach based on genetic programming. In Section 4, we validate our solution by reporting an empirical experiment we conducted. In Section 5, we discuss the application and limitations of our approach. Finally, we review related work in Section 6 and conclude in Section 7.

2 CHALLENGES IN LEARNING REFACTORING TRANSFORMATIONS

2.1 Motivating example

Although there are well-documented refactoring patterns defined for known formalisms, e.g., UML class diagrams (UMLCD) [10], it is very difficult for a domain expert to express complete general refactoring rules for a DSL. Nevertheless, such non-software engineering experts typically provide their refactoring knowledge by means of examples. However, learning general rules from examples is not trivial, since it is very sensitive to the coverage of the examples.

Consider the *Pull-up field* refactoring pattern in UMLCD. It is usually described as *"if two subclasses have the same field, move that field to their super-class"* [10]. To learn the rule for this refactoring as a model transformation, the domain expert could provide the example illustrated in Figure 1 (a). From this example, one could easily deduce a single model transformation rule where the precondition pattern, a.k.a. left-hand side (LHS), consists of a super-class C_1 with two sub-classes C_2 and C_3 having both an attribute A_1. This pattern is a generalization of the subset of the left model with classes `A`, `B` and `C` and attributes `x`. The action part of the rule, a.k.a. the right-hand side (RHS), consists of deleting A_1 from C_2 and C_3, and creating an attribute A_1 in C_1.

An important challenge with learning this rule from this example is that the algorithm should detect the part that has been modified so that it correctly identifies opportunities to apply this refactoring on any input UMLCD model. Existing approaches learning model transformation from examples cannot be reused for learning refactoring rules. Some of them [1, 9] try to learn outplace transformations, where a new model is produced from another one. Others [12, 32] try to only learn the sequence of refactoring rule application given the rules and input model, from example. For a refactoring, the model transformations to learn are inplace: it is the same model that is modified [29]. For such transformations, the challenge is to detect the transformation occurrence correctly in the model, rather than focusing on the changes to perform. **This is therefore a key contribution of this article.** In the previous example, the rule must detect that two subclasses need to be present from the model pair in Figure 1 (a). That is because a counterexample is also present between classes `F` and `G`. This helps to reduce the search space for finding the correct precondition to apply a refactoring.

Another challenge is that the rule deduced from solely this example only works when there are two subclasses. If there are three or more subclasses as in Figure 1 (b), applying this rule will only remove `x` from two subclasses. A better model transformation for this refactoring would consist of two rules. A first rule duplicates an attribute in the super-class when this attribute exists in two sub-classes. A second rule removes an attribute from a sub-class when this attribute exists in the super-class. **This is in fact the transformation output by our algorithm.** Nevertheless, this revised transformation still has corner cases when it is erroneous. For example, if the model to refactor is the one in the left part of Figure 1 (c), this transformation will pull attribute `x` up to `A` even if it should not be defined on all its subclasses (i.e., `D`). In this case, the correct model transformation should pull the common attribute to a new intermediate subclass, leading to the desired refactored model illustrated on the right of Figure 1 (c).

This example illustrates that even for commonly known refactoring patterns in software engineering, manually expressing the general model transformation that implements a refactoring pattern is very difficult. It is even more difficult for domain-specific experts who are not used to think algorithmically, such as in [6]. It also shows how critical the appropriate choice of examples is to learn the right refactoring rules.

2.2 Problem formulation

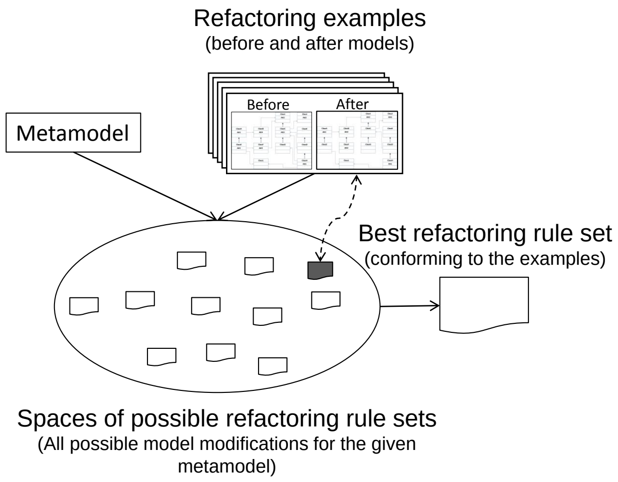
Refactoring examples
(before and after models)
Before
After
Metamodel
Best refactoring rule set
(conforming to the examples)
Spaces of possible refactoring rule sets
(All possible model modifications for the given metamodel)

Figure 2: Refactoring learning as an optimization problem

Consider the situation where no explicit refactoring knowledge is available. The only available information is the DSL described by means of a metamodel and a set of examples each containing a pair of models, i.e., before and after the refactoring, as depicted in Figure 2. In this situation, learning refactoring rules can be seen a search problem in which we explore the very large space of all possible refactoring rule sets that can be written for the concerned metamodel. To guide the search, candidate rule sets are evaluated using the conformance of their behavior with the provided examples, i.e., applying the refactoring rules to the before models of the example pairs and comparing the obtained models with the corresponding after models.

3 LEARNING REFACTORING TRANSFORMATIONS FROM EXAMPLES

In this section, we start by presenting the basics of genetic programming. Then, we show how we adapt this algorithm to the problem of refactoring rule learning.

3.1 Genetic programming

Genetic programming (GP) is an evolutionary algorithm whose goal is to automatically create computer programs, from examples of inputs/outputs, to solve problems [28]. Figure 3 sketches its general process. The algorithm starts by generating an initial population of solutions (programs) of a given size. Then, it evaluates these solutions by means of a fitness function. This function generally measures the ability of a program to produce the expected outputs from the provided inputs. The next step of the algorithm is to evolve the initial population through a given number of iterations (generations) by combining three types of operations: elitism, crossover, and mutation. Thus, for each generation, a fixed number of the top solutions are automatically reproduced in the new population following the elitism principle. This principle ensures that the best solutions are not lost during the evolution. Then, to fill the remaining slots in the population, pairs of parent solutions are selected following a strategy that favors the fittest solutions, while still giving a chance to all the solutions. An example of such strategies is the roulette-wheel selection that consists in assigning a probability to each solution to be selected, proportionally to its fitness. Each selected pair of parent solutions is used, with a certain probability, to produce two child solutions, thanks to the crossover operator. The child solutions (or parent solutions if the crossover is not performed) are mutated with a given probability. When a stop condition is satisfied (generally a fixed number of generations), GP returns the best solution found.

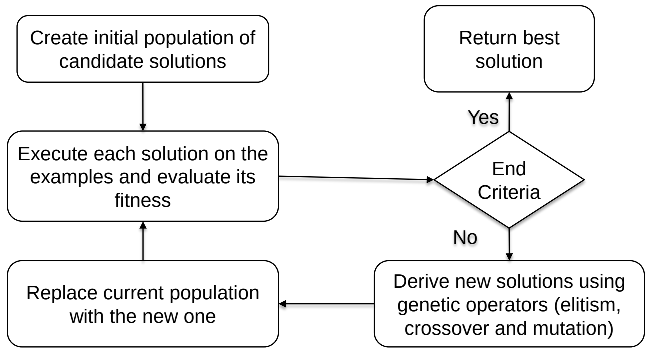
Create initial population of candidate solutions
Return best solution
Yes
Execute each solution on the examples and evaluate its fitness
End Criteria
No
Replace current population with the new one
Derive new solutions using genetic operators (elitism, crossover and mutation)

Figure 3: Overview of the genetic programming process

The above-described algorithm is generic and can be applied to derive any kind of program. To adapt the algorithm to a specific problem, we shall define: (1) how to encode a program (in our case a set of refactoring rules), (2) how to generate an initial population of programs, (3) how to evaluate a program, and (4) how to produce new programs from existing ones using genetic operators. The remainder of this section details these specifics to learn refactoring rules from examples.

3.2 Encoding refactoring rule sets

To learn the refactoring rules, we have a set of m examples, where each example i consists of a source model s_i and a target model e_i. The task is to find all refactoring rules that transform any model s_i into e_i. Therefore, a candidate solution is a set of refactoring rules, which we call *refactoring set* in the following. We denote each individual transformation by $R = \{r_1, r_2, \ldots, r_k\}$ for $k \in \mathbb{N}$, composed of k refactoring rules r to be executed. Note that a set of rules R encodes a set of refactorings because the provided example pairs may correspond to more than one refactoring pattern application. The initial population is therefore composed of n sets of refactoring rules.

```
1  (defrule PullUpField
2    ?c1 <- (Class(name ?A))
3    ?c2 <- (Class(name ?B))
4    ?c3 <- (Class(name ?C))
5    ?i1 <- (Inheritance(subclass ?B)(superclass ?A))
6    ?i2 <- (Inheritance(subclass ?C)(superclass ?A))
7    ?a1 <- (Attribute(name ?attName)(class ?B))
8    ?a2 <- (Attribute(name ?attName)(class ?C))
9    (test (and (neq ?c1 ?c2)(neq ?c1 ?c3)(neq ?c2 ?c3)))
10   =>
11   (assert (Attribute(name ?attName)(classe ?A)))
12   (retract ?a1)
13   (retract ?a2))
```

Listing 1: A rule encoded in the Jess languagee

To encode a refactoring rule, we use the general purpose declarative language Jess (Java Expert System Shell) [13]. We specify a metamodel by a set of *fact templates*, by textually encoding the metamodel elements, such as classes and inheritance relationships for UMLCD. We encode models as *facts*, instances of the fact templates, e.g., `class(name Student)` and `inheritance(subclass Student)(superclass Person)`. The example given in Listing 1 shows a candidate rule for the *Pull-up field* refactoring[1]. The LHS of the rule encodes a refacotring opportunity, i.e., a pattern to search for in the source models (lines 2–9 in our example). The RHS of the rule lists the sequence of operation to perform on the LHS pattern instances (lines 11–13). The pattern to match includes three classes (lines 2–4). Two of them should inherit from the third one (lines 5–6). The two subclasses should also have an identical attribute (lines 7–8). When an instance of this pattern is found, the operation to perform are: add an attribute in the superclass with the same name as the ones of the subclasses (line 11) and remove the attribute from the subclasses (lines 12–13). We can see that this rule will correctly detect the refactoring in the example pair of Figure 1 (a).

[1] Note that this is a simplified rule, types cardinalities and other properties are also taken into account.

3.3 Initial creation of refactoring sets

As shown in Figure 3, the first step is to create randomly an initial population of n refactoring sets. Each refactoring set contains a random number of rules within a parameterized interval. The LHS of a rule is created by generating randomly a bounded number of fragments based on the metamodel types. The state conditions on the fragment are generated randomly based on the element properties as described in [9].

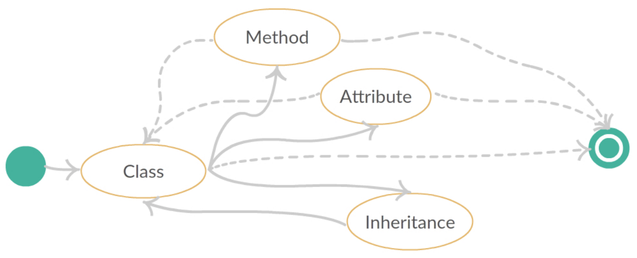

Figure 4: FTG to create fragments of a excerpt of the Class Diagram metamodel

To help creating consistent (strongly connected) patterns to search for, we use a strategy based on the constraints imposed by the metamodel structure. To this end, we build a fragment type graph (FTG), a sort of automaton that defines the dynamics of the pattern elements' creation. Figure 4 shows an excerpt of an FTG corresponding to a simplified UMLCD metamodel. In such a graph, nodes are either element types of the metamodel to instantiate or the start/end nodes. When traversing this graph, the current node indicates the metamodel element from which we can create the next element. These nodes are connected by two types of edges. A creation edge (depicted as a solid arrow) indicates that the target node can be created after the source node is created. Both created elements are connected according to the metamodel syntax. For example, the edge between the *Class* and *Attribute* means when a class $C1$ is created, the next step may be the creation of an attribute $A1$. $A1$ is then considered as an attribute of $C1$. The second type of edge is the back edge (dashed arrow). It indicates that from the current element, we cannot further create a metamodel element connected to it. For example, once we create the attribute $A1$, there is no further element to create that can be connected to it. The back edge outgoing from $A1$ sets $C1$ as the following element. When a node is the source of many creation edges, one of the edges is selected randomly. For example, after setting $C1$ as the current node, it is possible to create a method $M1$ for $C1$, another attribute $A2$ for $C1$ or an inheritance relationship $I1$ with $C1$ as the superclass or the subclass. In the latter case, the current instance $I1$ requires the creation of a new class $C2$ to play the role of the superclass. $C2$ becomes the current instance and so on and so forth. When a node has many back edges, then one of them is selected arbitrary. If we select the edge whose target is the end node, the pattern creation stops and this constitute the LHS of the rule.

To create the RHS of rules, we randomly select addition and deletion operators with randomly defined elements.

3.4 Evaluating refactoring sets

We determine how well a set of refactoring rules R implements the transformation of the m examples provided using a global fitness function F. As shown in Equation (1), it averages the fitness functions $f_i(R)$ that calculate how "close" R is from producing the expected result for each example.

$$F(R) = \frac{\sum_{i=1}^{m} f_i(R)}{m} \tag{1}$$

Like other approaches that learn transformations [1, 2, 8, 9], to evaluate $f_i(R)$, we apply R on the considered source model s_i of the example pair i to produce a model p_i that we compare with e_i. However, because we deal with inplace transformations, we must ensure that R only modifies the concerned model elements and preserves the others. To define f_i, let us consider the following four sets. A and D are the sets of elements added and deleted by R respectively, when comparing s_i with p_i. EA and ED are the sets of elements expected to be added and deleted respectively, when comparing s_i with e_i. SD is the set of elements that should not be deleted, when comparing s_i, p_i and e_i. Then, we compute the following sets. $CA = A \cap EA$ and $CD = D \cap ED$ are the sets of elements that have been correctly added and deleted respectively. $IA = A \setminus EA$ and $ID = D \setminus ED$ are the sets of elements that have been incorrectly added and deleted respectively. To avoid bias favoring rules with higher number of elements, these sets are computed for each type $t \in T_i$, T_i being the set of types present in the metamodel and instantiated in example i.

The fitness function $f_i(R)$ includes for each type t two components, $mod_t(R)$ and $pres_t(R)$ that calculate respectively the model modification and preservation scores when applying R to s_i[2]:

$$f_i = \frac{1}{|T_i|} \times \sum_{t \in T_i} \alpha \times mod_t + \beta \times pres_t \tag{2}$$

$mod_t(R)$ is the proportion of elements of type t added or deleted by R among the expected ones. If no change is expected for a type t, then $mod_t(R) = 1$. Formally:

$$mod_t = \begin{cases} 1, & \text{if } |EA_t| + |ED_t| = 0 \\ \frac{|CA_t| + |CD_t|}{|EA_t| + |ED_t|}, & \text{otherwise} \end{cases} \tag{3}$$

$pres_t(R)$ is the proportion of elements in s_{it} (elements of type t) that are not supposed to change and that are actually preserved. It is defined formally as:

$$pres_t = \frac{1}{2} \times \left(\left(1 - \frac{|IA_t|}{|s_{it}| + |IA_t|} \right) + \left(1 - \frac{|ID_t|}{|SD_i|} \right) \right) \tag{4}$$

[2]To make the notation less cluttered, we omit "(R)" in the various components of f_i equations.

3.5 Deriving new refactoring sets

To generate the next population of rule sets, we start by selecting a given number of the fittest ones and include them directly in the generated population. This elitism strategy allows us to preserve the best genetic material across generations. Then, to fill the remaining slots, we perform the crossover and mutation operators on the rule sets of this generation. More specifically, we select a pair of rule sets following the roulette-wheel strategy. This assigns to each rule set a probability to be selected proportionally to its fitness. When two parent rule sets are selected three equiprobable scenarios can be performed: crossover only, mutation only, or both. Whatever the chosen scenario is, the crossover and mutation operations are performed with a certain probability.

Crossover. The crossover operator produces two new rule sets by combining the rules of the parent rule sets. Let us consider the two rule sets R_a and R_b. If the crossover is chosen, we apply it with a certain probability as follows. *Cut-points*, x and y are respectively selected to partition each set of rules into two: $R_a = \{r_{a1}, \ldots, r_{ax}, r_{ax+1}, \ldots, r_{ak}\}$ and $R_b = \{r_{b1}, \ldots, r_{by}, r_{by+1}, \ldots, r_{bl}\}$. The we recombine the partitions to produce two new rule sets: $R_{a'} = \{r_{a1}, \ldots, r_{ax}, r_{by+1}, \ldots, r_{bl}\}$ and $R_{b'} = \{r_{b1}, \ldots, r_{by}, r_{ax+1}, \ldots, r_{ak}\}$.

Mutation. If mutation is chosen for a given rule set, one of four mutation operators modifies the rules themselves. At the rule set level, an operator adds/removes a randomly generated/selected rule (as described in Section 3.3). At the rule level, an operator adds/removes pattern elements in the LHS, while still conforming to the FTG. Finally, two operators adds/removes modification operations in the RHS: one for assertion and one for deletion operations (see Section 3.2). To avoid any bias, all operators are selected with equal probability. However, a great part of the problem complexity is to search for the accurate refactoring opportunity before performing the refactoring. Therefore, we assign the LHS mutation operator twice as much probability as the others.

4 VALIDATION

We evaluate our approach along the following three research questions:

- **RQ1 Do the learned refactoring sets refactor models correctly?** We first verify that all the refactorings have been correctly applied, such that the discovered transformation rules produce the correct target models.
- **RQ2 Do the expected refactorings appear in the learned rules?** We then validate that the transformation rules correctly implement the intended refactorings from the examples. This guarantees that we do not achieve the production of the correct target model resulting from an arbitrary combination of rules, but thanks to the correct refactoring rules.
- **RQ3 Are the results obtained attributable to our approach?** Since we rely on a probabilistic evolutionary approach, we have to check if we consistently obtain similar results across executions, and if these results

Table 1: Selected metamodels, refactorings, and model examples

Metamodel	Refactoring	Description	Occurrences	Models
UMLCD	Pull up field	Move an attribute shared by two classes to a superclass	3	5
	Pull up method	Declare the method in the superclass and keep the method definition in the current class	4	
	Pull up association	Move an association shared by two classes to a superclass	1	
	Clean up attribute	Delete an attribute of a sub-class if it is also defined in the superclass	4	
WPN	Removing implicit place	Remove a place that does not change the overall marking of the net	2	4
	Removing EFC structures	If a set of places all have arcs to the same set of transitions, introduce an intermediate transition and place to direct the token flow from the set of places to the set of transitions	3	
	Removing TP-cross structures	If a set of transitions all have arcs to the same set of places, introduce an intermediate place and transition to direct the token flow from the set of transitions to the set of places	1	
BPMN	Pull up incoming sequence flow	If a sequence flow connects two tasks where one is in a process and the other in a sub-process, the flow should connect the task to the sub-process directly	2	3
	Pull up outgoing sequence flow	Similar as above, but from the sub-process to the task	2	
	Replace sequence by message flow	Replace a sequence flow between two classes in different pools by a message flow	2	
	Explicit data association	If a sequence flow connects two tasks via an artifact, connect the two tasks directly with a sequence flow and use a data association for the artifact	2	
	Symmetric modeling	Every nested opening gateway must have its corresponding closing gateway in the same order	1	

are better than those of the best solution found by a random exploration which considers an equal number of solutions.

4.1 Experiment setting

This section details the evaluation setting. This includes the considered metamodels and their respective refactorings, the examples used in the learning process, the parameters of the GP algorithm, and the method used to answer the research questions.

4.1.1 Data collection. To test our approach, we use three metamodels, for which the refactoring rules are known (ground truth) and there are sufficient data sources to collect examples from. For each metamodel, we collected examples as pairs of before and after models available online or in the literature. Often, the model examples include multiple refactorings. Table 1 summarizes the data used. For each metamodel, we list the refactoring types performed in the examples, the number of refactoring-type occurrences found in the examples, and number of models considered.

UML class diagrams (UMLCD). For UMLCD, we reverse engineered partial models from two well-known open-source Java projects: *ArgoUML* and *Xerces-J*. Since these projects have many releases available, we selected model versions before and after refactoring. Using XSLT, we implemented templates to generate sets of facts in Jess from the XML models.

From the given examples, we identified the four refactorings[3] reported in Table 1 as cataloged in [10]. It is interesting to note that some refactorings can overlap. For example, *Clean up attribute* is also part of *Pull up field*. We anticipate that their respective transformations will share a rule.

[3]It is not necessarily a refactoring per se, but a form of anomaly correction.

Workflow Petri Nets (WPN). WPN is a particular class of Petri nets with a single source place, a single sink place, and all places and transitions are on a path from the source to the sink. In [35], the authors formalize three refactorings that improve the execution of the WPN by removing redundant elements in a way that does not change the observable behavior of the net. The refactoring are complex to implement as the authors provide several algorithms to apply them. In [35], they also provide four examples of WPN.

Business Process Model and Notation (BPMN). BPMN is a widely used formalism for business analysts to create, implement, and monitor processes [26]. Several works have proposed refactoring opportunities to improve the semantics and readability of BPMN models. The work in [27] provides four good and bad practices related to task connectivity. We also considered an additional refactoring called *Symmetric modeling* from [5]. We collected three models from the OMG standard [26] to detect these refactoring opportunities.

4.1.2 Algorithm parameters. Like for any evolutionary algorithm, one needs to specify the parameters to run our GP-based approach. To tune these parameters, we ran our approach with different configurations on different data sets. For the validation, we retained the following configuration. We set the size of the initial populations to 30 candidate refactoring rule sets. This size is constant for all evolutions of the population. We set both probabilities of crossover and mutation to 0.9. Unlike classical genetic algorithms, having a high mutation probability is not unusual for GP algorithms (see for instance [30]). For the elitism, we injected the top 3 refactoring rule sets into the next generation.

Table 2: Results of learned refactorings with GP

Metamodel	Run	Fitness	Precision	Recall	Rules obtained by GP			Expected rules
					Exact	Partial	Unexpected	
UMLCD	1	100%	100%	100%	3	1	0	4
	2	100%	100%	100%	3	1	0	
	3	100%	100%	100%	3	1	0	
WPN	1	88.7%	98.0%	77.2%	1	3	0	3
	2	82.0%	100%	83.8%	2	1	1	
	3	82.0%	100%	74.5%	2	2	0	
BPMN	1	98.8%	72.6%	89.3%	2	2	2	5
	2	98.8%	72.6%	89.3%	2	1	3	
	3	99.4%	81.0%	94.8%	3	2	1	

4.1.3 Methodology. To answer RQ1, we compare the produced model p_i with the expected model e_i, relying on precision and recall measures. In Equation (5), we define precision as the ratio between number of correct modifications and the total number of modifications, based on the sets defined in Section 3.

$$Precision = \frac{1}{|T_i|} \times \sum_{t \in T_i} \frac{|CA_t| + |CD_t|}{|CA_t| + |CD_t| + |IA_t| + |ID_t|} \quad (5)$$

In Equation (6), we define the recall as the ratio between number of correct modifications and total number of expected modifications, which we defined as *mod* in Equation (3).

$$Recall = \frac{1}{|T_i|} \times \sum_{t \in T_i} \frac{|CA_t| + |CD_t|}{|EA_t| + |ED_t|} \quad (6)$$

For RQ2, we focus on the quality of the refactoring rules obtained with respect to the known refactoring rules. We manually inspect the best set of refactoring rules obtained for each metamodel to determine how close they are to the expected ones. We determine if an obtained rule completely or partially matches the expected one, or if it is not expected at all. We also check if some expected rules were missing.

To answer RQ3, we perform five runs of our approach for UMLCD with $10,000$ generations. We also use five runs of random generation, each having 30 000 rule sets (equivalent to 30 solutions per generation $\times$ 10 000 generation). We generate the random rule sets according to the initial population generation procedure. Then for each run (GP and random), we select the solution with the best fitness. We compare our approach with the random generation in terms of precision and recall of the obtained rule sets.

4.2 Results and interpretation

4.2.1 RQ1: Correct refactored models. The left part of Table 2 presents the results of the fitness, precision, and recall scores that report on the quality of the refactored models. For each metamodel, we report the results of three executions as the learning process is probabilistic by nature. The three executions use exactly the same settings, including the same initial population. For UMLCD, we observe perfect results on three executions for both precision and recall when comparing the refactored models generated by the learned rules with those given in the examples.

The precision in the case of WPN is also almost perfect. Except for one execution where a slight modification was considered as incorrect, all additions and deletions of model elements were appropriate. However, the recall is relatively low (between 74% and 84%), meaning that some modifications where not performed as expected. After analyzing the obtained models, this recall score can be explained by the following considerations. The WPN metamodel contains few elements, essentially places, transitions, and arcs. Then, the sought refactoring are complex variations of instances of the same few element types. For example, a refactoring rule removes an unnecessary place depending on other places and transitions. To learn such a rule, the provided before- and after-refactoring examples should include enough fragments differentiating this situation from other similar situations where the refactoring is not necessary.

For the BPMN metamodel, we observe an opposite tendency, i.e., a better recall and a lower precision. The lower precision can be explained by the fact that some additions and deletions were made because the learned rules missed some conditions to have a more precise pattern to search for.

4.2.2 RQ2: Correct refactoring rules. Producing the correct models using the learned rules does not mean that these rules are those expected. For this research question, we investigate whether the learned rules are correct with respect to the known and expected refactorings. Note that a specific type of refactoring can be realized by means of one or more refactoring rules. Thus we will not have one-to-one comparisons between the learned and expect rules. We rather decide if a learned rule has a correct contribution to a refactoring alone or in conjunction with other rules.

The right part of Table 2 presents an overview of the rule sets obtained by our approach compared to the expected one. For the UMLCD metamodel, we were expecting four refactoring types (see Table 1). We obtained the exact rule for the *Clean up attribute*. The combinations of this rule with respectively two other rules implement exactly the *Pull up field* and *Pull up method* refactorings. We have the same

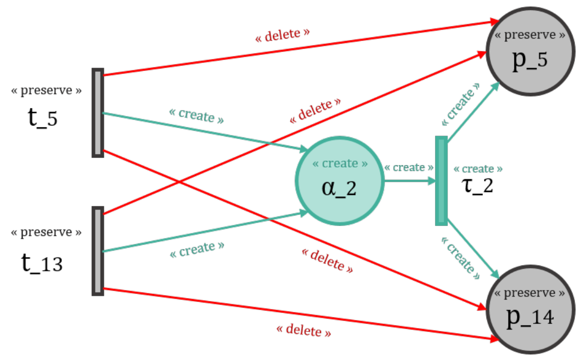

Figure 5: Rule obtained by GP for the *Removing TP-cross structures* refactoring

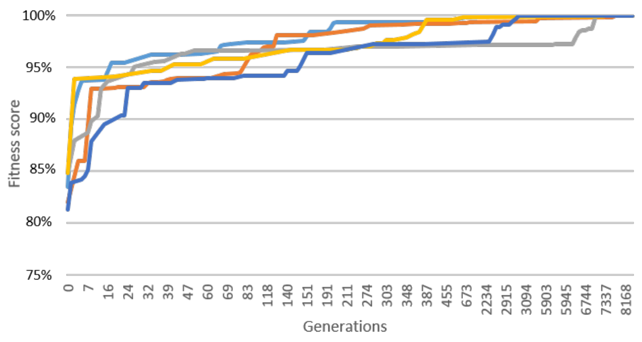

Figure 6: Evolution of fitness score for UMLCD over five runs using our approach

situation with *Pull up association* with the exception that one condition was missing in one rule, i.e., checking that an inheritance relationship exists only for one of the two classes having the association to pull up. This missing condition did not affect the precision and recall in RQ1, because, there was no counterexample in the provided models where a missing inheritance prevents from performing the refactoring.

For the WPN metamodel, we were able to learn all three refactorings. However, the GP consistently found, for the three executions, one more rule. In one execution, this additional rule has nothing to do with the expected refactorings. In the other cases, the additional rules contribute partially to the refactorings together with the other rules. The learned rules are not trivial. Figure 5 illustrates the rule we obtained for the *Removing TP-cross structure* refactoring. It looks for two transitions, each connected to the same two places and then replaces these arcs with a place and a transition to connect the initial pair of transitions to the initial pair of places.

For the BPMN metamodel, the GP also generated more rules than expected. But in this case, many of these rules were not contributing to the expected refactoring: this explains the relatively low precision score for RQ1. We accurately obtained the expected rules for two refactorings, namely *Symmetric modeling* and *Explicit data association*. For the three other refactorings, the GP was not able to derive the complete refactoring rules. As illustrated in Table 2, it only partially matched some of the rules and missed others. This situation illustrates the fact that the derived rules do not necessarily have the exact same structure (pattern elements and relations) as the rules we would have written by hand. Nevertheless, the learning process produces rules that, overall correctly perform the refactoring with respect to the provided examples.

4.2.3 RQ3: GP vs random. Before comparing the precision and recall of the learned refactoring rule sets, we first look at the convergence of the five GP executions. Figure 6 shows the curves corresponding to the best fitness evolution during the respective executions. These curves always converge towards a fitness score of 100% (as reported in Table 2) after a certain number of generations.

To compare with the random generation, we obtained perfect precision and recall for all the five executions of GP. As Table 3 shows, the precision is below 51% and the recall is below 56% for the five random executions. We can conclude that the results obtained for UMLCD are attributable to our learning algorithm and not to the number of explored solutions.

Table 3: Results of random generation for UMLCD

Run	Fitness	Precision	Recall
1	85.3%	50.7%	55.4%
2	86.6%	26.9%	42.1%
3	86.3%	45.0%	49.8%
4	86.3%	42.0%	45.0%
5	86.6%	48.6%	48.8%

4.3 Threats to validity

The evaluation of our approach has shown that it can learn complex refactorings in many cases. However, a threat to the validity of these results is the generalization of these results to DSLs, as mentioned in our motivations. Although experimenting with such DSLs is important, the need of having known refactorings (ground truth) and available model examples, limits our possibilities. Still, we believe that the considered metamodels and refactorings are complex enough and exhibit a good variety to represent an acceptable setting for a preliminary evaluation.

The choice of the examples is another possible threat. We selected examples that are almost prototypic. In real-life scenarios, models can be modified for different purposes, so it is not always possible to distinguish refactorings from other changes. However, our setting matches the scenario where a modeler starts performing the refactoring on some fragments and then gives the initial and modified fragments to our learning algorithm.

Another validity threat is concerned with the stop criterion for the GP algorithm. We ran our algorithm for a fixed amount of time (hours) and then checked how many generations were needed to find the best rule set. In a real-life scenario, waiting for such an amount of time is not always feasible, especially if the modeler uses our approach in an interactive way.

5 DISCUSSION AND LIMITATIONS

Like many other example-based approaches, the quality of the learned rules strongly depends on the quality of the examples provided. As we have shown in the BPMN case, it is utterly important to provide examples having enough variations to distinguish when different, but close, situations may lead (examples) or not (counterexamples) to a refactoring.

Related to this observation, we also noticed that the partially matched rules were under or over-constrained. Many of these situations could have been avoided had we integrated negative conditions. Our rule creation procedure generates conditions that check for the presence of elements instantiating a pattern. However, in some cases, one has to also check for the absence of elements. Integrating negative conditions in the learning process will increase the expressiveness of the learned rules.

In our current implementation, we are able to successfully detect structural anomalies in the model to be refactored, i.e., related to the presence of metamodel element instances and their connectivity. We have found cases in WPN, where the precondition of a rule requires universal quantification (e.g., *Removing TP-cross structures*). In a by-example approach, it is hard to generalize a rule from a finite sample set of examples. As in Figure 5, the GP was only able to find rules for two and three transitions, because the examples did not contain situations with more transitions. However, the general rule in [35] states that it applies to an arbitrary number of transitions. This also comes from the limitation of the model transformation paradigm that relies on existential quantification. It is, nevertheless, possible in Jess to add queries and functions in the LHS to generalize the matching. Such queries can be implemented according to navigation possibilities for a given metamodel, independently from the sought refactorings. For example, in UMLCD, a primitive query can consist of generating the set of the associations involving a given class. During the learning, this set can be used to test the absence of associations (size equals 0, for example). This approach was successfully used in [1].

Other types of refactorings are not structural by nature. For instance, in UMLCD, a pair of models example may exhibit shortening an attribute with a very long name. The intent behind this semantic anomaly correction is not only very hard to encode, but also hard to derive from a by-example approach. As this refactoring is not related to the model structure, generating the rule conditions (LHS) is not possible in the current version. To handle this, string and arithmetic operations must be used, as done in [1].

6 RELATED WORK

In the literature, many tools have been proposed to refactor models [17, 22, 25, 29, 31, 37]. Although they use model transformation to refactor models, they do not learn refactorings automatically. In this section, we discuss approaches that automate model refactoring and approaches that derive model transformations from examples or demonstrations.

6.1 Search-based model refactoring

Our approach is a search-based technique and, like very few others, is dedicated to refactoring models [21]. Most search techniques rely on evolutionary algorithms, especially genetic algorithms. In [12], the authors also propose to generate a sequence of refactoring by measuring the similarities between provided examples. Instead, our fitness function measures explicitly correct and incorrect modifications. SORMASA [3] is a tool that assists the user by suggesting a set of model refactorings. It relies on a mono-objective evolutionary algorithm aiming at increasing cohesion and reducing coupling of UMLCD models. In [14], the author present an approach that attempts to introduce design patterns in UMLCD models by optimizing specific software design metrics.

There are, nevertheless, some approaches that consider behavior preservation. For example in [20], the authors implement a multi-objective evolutionary algorithm to optimize the trade-off between improving the quality related UML models.

6.2 Model transformations by demonstration

By demonstration approaches are used to derive inplace transformations automatically. In [4], the authors proposed an approach to alleviate the complexity of developing model refactoring operations. They derive semi-automatically endogenous model transformations by analyzing user editing actions when refactoring models. They collect all atomic operations performed by the user by comparing the initial and final models. The operations are then saved in a difference model from which they propose a set of pre/postconditions of the refactoring operations to the user for manual refinement. Sun et al. [33] propose a similar approach for deriving similar transformations. To collect the operations, they extend the modeling environment to monitor the user editing actions. The recorded operations are then analyzed to remove incoherent and unnecessary ones. Using an inference engine, they express user intentions as reusable transformation patterns. In [15], the authors proposed an automatic inference of inplace transformations. Their approach can infer advanced rule features, such as negative application conditions, multi-object patterns, and global invariants.

6.3 Model transformation by examples

Several works have investigated how to learning model transformations from examples [7, 36, 38]. However, these approaches are limited to exogenous and outplace model-to-model transformations. Learning inplace transformations from examples is more complex because we must preserve the unaffected elements of the model. Faunes et al. [8, 9] used GP learn transformation rules from examples of outplace transformations. Two articles [1, 16] tried to address more complex transformations. Although the latter did not need to learn rules, the former employed a strategy based on dividing the learning process in different steps, using adaptive mutation, and using fine-grained transformation traces in the examples.

7 CONCLUSION

In this paper, we present an approach to recommend refactoring rules learned from examples. The examples are pairs of models representing the versions before and after the application of refactorings. The learning is performed using GP by evolving a population of randomly generated sets of model transformation rules guided by the conformance to the provided examples. As such our approach does not pretend to find the absolute refactoring rules for a given modeling language. It instead finds the rules that best conform to the considered examples.

To evaluate our approach, we applied it on three metamodels together with refactored model examples. This evaluation showed that our approach can learn complex refactoring rules and that the obtained results are not attributable to the number of explored solution, but to our search strategy.

Although the obtained results are satisfactory, there is room for improvement. First, expressiveness of the learned rules can be enhanced by considering new constructs such as negative conditions, navigation primitives, and arithmetic and string operations. The performance of our algorithm in terms of execution time deserves to be improved. We plan to optimize the best set of rules found by the GP to improve the efficiency of the rule engine. Finally, the accuracy of our approach can be enhanced by considering more sophisticated genetic operators and search strategies. For example, adaptation mutation can be used to adapt the search strategy to the characteristics of the current population, as in [1]. This includes changing the mutation probability when no improvement is noticed for many generations or if the diversity inside the populations decreases. Another way of improving the search strategy is by experimenting with multi-objective GP algorithms. The *mod* and *pres* scores can be implemented as two different objectives without the need of defining weights to aggregate them. Minimizing the size of the rules can be another objective to avoid rules with unnecessary conditions. In the future, we also plan to test our approach on a larger set of metamodel, including DSLs, for which it is difficult to define refactoring rules.

REFERENCES

[1] Islem Baki and Houari Sahraoui. 2016. Multi-Step Learning and Adaptive Search for Learning Complex Model Transformations from Examples. *ACM Transactions on Software Engineering Methodology* 25, 3 (2016), 20:1–20:37.

[2] Islem Baki, Houari Sahraoui, Quentin Cobbaert, Philippe Masson, and Martin Faunes. 2014. Learning Implicit and Explicit Control in Model Transformations by Example. In *Model-Driven Engineering Languages and Systems*. Springer, 636–652.

[3] Thierry Bodhuin, Gerardo Canfora, and Luigi Troiano. 2007. SORMASA: A tool for Suggesting Model Refactoring Actions by Metrics-led Genetic Algorithm. In *Workshop on Refactoring Tools in conjunction with ECOOP*. 23–24.

[4] Petra Brosch, Philip Langer, Martina Seidl, and Manuel Wimmer. 2009. Towards end-user adaptable model versioning: The by-example operation recorder. In *ICSE Workshop on Comparison and Versioning of Software Models*. IEEE Computer Society, 55–60.

[5] Camunda. 2018. BPMN Examples–Best Practices for creating BPMN 2.0 process diagrams. https://camunda.com/bpmn/examples/. (2018). (last accessed: apr 2018).

[6] J. Cunha, J. P. Fernandes, P. Martins, R. Pereira, and J. Saraiva. 2014. Refactoring Meets Model-Driven Spreadsheet Evolution. In *Quality of Information and Communications Technology*. 196–201.

[7] Xavier Dolques, Marianne Huchard, Clémentine Nebut, and Philippe Reitz. 2010. Learning Transformation Rules from Transformation Examples: An Approach Based on Relational Concept Analysis. In *Enterprise Distributed Object Computing Workshops*. 27–32.

[8] Martin Faunes, Houari Sahraoui, and Mounir Boukadoum. 2012. Generating Model Transformation Rules from Examples using an Evolutionary Algorithm. In *Automated Software Engineering*. 1–4.

[9] Martin Faunes, Houari Sahraoui, and Mounir Boukadoum. 2013. Genetic-programming approach to learn model transformation rules from examples. In *Theory and Practice of Model Transformations (LNCS)*, Vol. 7909. Springer, 17–32.

[10] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[11] A. Garrido, G. Rossi, and D. Distante. 2007. Model Refactoring in Web Applications. In *IEEE International Workshop on Web Site Evolution*. 89–96.

[12] Adnane Ghannem, Ghizlane El-Boussaidi, and Marouane Kessentini. 2014. Model refactoring using examples: a search-based approach. *Journal of Software: Evolution and Process* 26, 7 (2014), 692–713.

[13] Ernest Friedman Hill. 2003. *Jess in Action: Java Rule-Based Systems*. Manning Greenwich, CT.

[14] Adam C. Jensen and Betty H. C. Cheng. 2010. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Genetic and Evolutionary Computation Conference*. 1341–1348.

[15] Timo Kehrer, Abdullah M. Alshanqiti, and Reiko Heckel. 2017. Automatic Inference of Rule-Based Specifications of Complex Inplace Model Transformations. In *Theory and Practice of Model Transformation*. 92–107.

[16] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Omar Ben Omar. 2012. Search-based model transformation by example. *Software & System Modeling* 11, 2 (2012), 209–226.

[17] Yasser A. Khan and Mohamed El Attar. 2016. Using model transformation to refactor use case models based on antipatterns. *Information Systems Frontiers* 18, 1 (2016), 171–204.

[18] S. Kolahdouz Rahimi, K. Lano, S. Pillay, J. Troya, and P. Van Gorp. 2014. Evaluation of model transformation approaches for model refactoring. *Science of Computer Programming* 85 (2014), 5–40.

[19] Levi Lúcio, Moussa Amrani, Juergen Dingel, Leen Lambers, Rick Salay, Gehan M.K. Selim, Eugene Syriani, and Manuel Wimmer. 2014. Model Transformation Intents and Their Properties. *Software & Systems Modeling* 15, 3 (2014), 685–705.

[20] Usman Mansoor, Marouane Kessentini, Manuel Wimmer, and Kalyanmoy Deb. 2017. Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. *Software Quality Journal* 25, 2 (2017), 473–501.

[21] Thainá Mariani and Silvia Regina Vergilio. 2017. A systematic review on search-based refactoring. *Information & Software*

Technology 83 (2017), 14–34.

[22] Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. 2009. Generic Model Refactorings. In *Model Driven Engineering Languages and Systems (LNCS)*. Springer, 628–643.

[23] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, Miguel A Fernandez, Bjørn Nordmoen, and Mathias Fritzsche. 2013. Where does model-driven engineering help? Experiences from three industrial cases. *Software & Systems Modeling* 12, 3 (2013), 619–639.

[24] Maddeh Mohamed, Mohamed Romdhani, and Khaled Ghédira. 2009. Classification of model refactoring approaches. *Journal of Object Technology* 8, 6 (2009), 121–126.

[25] Olaf Muliawan and Dirk Janssens. 2010. Model refactoring using MoTMoT. *International Journal on Software Tools for Technology Transfer* 12, 3-4 (2010), 201–209.

[26] Object Management Group 2011. *Business Process Model and Notation (BPMN)* (2 ed.). Object Management Group.

[27] Gregor Polancic. 2013. *Understanding BPMN Connections*. Technical Report. Orbus software.

[28] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. 2008. *A field guide to genetic programming*. Lulu Enterprises, UK Ltd.

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

[30] Sam Ratcliff, David Robert White, and John A. Clark. 2011. Searching for invariants using genetic programming and mutation testing. In *The Genetic and Evolutionary Computation Conference (GECCO)*.

[31] Jan Reimann, Mirko Seifert, and Uwe Aßmann. 2010. Role-Based Generic Model Refactoring. In *Model Driven Engineering Languages and Systems*. Springer, 78–92.

[32] Olaf Seng, Johannes Stammel, and David Burkhart. 2006. Search-based Determination of Refactorings for Improving the Class Structure of Object-oriented Systems. In *Annual Conference on Genetic and Evolutionary Computation (GECCO '06)*. ACM, 1909–1916.

[33] Yu Sun, Jeff Gray, and Jules White. 2011. MT-Scribe: an end-user approach to automate software model evolution. In *International Conference on Software Engineering*. ACM, 980–982.

[34] Mohammad Tanhaei, Jafar Habibi, and Seyed-Hassan Mirian Hosseinabadi. 2016. Automating feature model refactoring: A Model transformation approach. *Information and Software Technology* 80 (2016), 138–157.

[35] Ichiro Toyoshima, Shingo Yamaguchi, and Jia Zhang. 2015. A Refactoring Algorithm of Workflows Based on Petri Nets. In *International Congress on Advanced Applied Informatics*. IEEE, 79–84.

[36] Daniel Varró. 2006. Model Transformation by Example. In *Model Driven Engineering Languages and Systems*. Springer, 410–424.

[37] I. Verebi. 2015. A model-based approach to software refactoring. In *International Conference on Software Maintenance and Evolution*. 606–609.

[38] Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler. 2007. Towards Model Transformation Generation By-Example. In *Annual Hawaii International Conference on System Sciences*. 285–295.