# Generating Efficient Mutation Operators for Search-Based Model-Driven Engineering

Daniel Strüber

University of Koblenz and Landau, Germany
strueber@informatik.uni-marburg.de

**Abstract.** Software engineers are frequently faced with tasks that can be expressed as optimization problems. To support them with automation, *search-based model-driven engineering* combines the abstraction power of models with the versatility of meta-heuristic search algorithms. While current approaches in this area use genetic algorithms with fixed mutation operators to explore the solution space, the efficiency of these operators may heavily depend on the problem at hand. In this work, we propose *FitnessStudio*, a technique for generating efficient problem-tailored mutation operators automatically based on a two-tier framework. The lower tier is a regular meta-heuristic search whose mutation operator is "trained" by an upper-tier search using a higher-order model transformation. We implemented this framework using the Henshin transformation language and evaluated it in a benchmark case, where the generated mutation operators enabled an improvement to the state of the art in terms of result quality, without sacrificing performance.

## 1 Introduction

Optimization lies at the heart of many software engineering tasks, including the definition of system architectures, the scheduling of test cases, and the analysis of quality trade-offs. Search-based software engineering (SBSE, [1]) studies the application of meta-heuristic techniques to such tasks. In this area, genetic algorithms have shown to be a particularly versatile foundation: the considered problem is represented as a search over solution candidates that are modified using mutation and crossover operators and evaluated using fitness criteria.

Combining SBSE with model-driven engineering (MDE), which aims to improve the productivity during software engineering via the use of models, is a promising research avenue: SBSE techniques can be directly applied to existing MDE solutions, mitigating the cost of devising problem representations from scratch. Success stories for transformation orchestration [2], version management [3], and remodularization [4] indicate the potential impact of such efforts.

In this context, we focus on a scenario where the task is to optimize a given model towards a fitness function. The state of the art supports this task with two classes of techniques: those involving a dedicated encoding of models [5,6], and those using models directly as solution candidates [7]. In both cases, applications typically involve a minimal set of transformation rules that is sufficient to modify

the solution candidates systematically, so that optima in the solution space can eventually be located. Yet, such modification does not put an emphasis on *efficiency* in the sense that improved solutions are located fast and reliably.

**Example.** This example is inspired by the 2016 Transformation Tool Contest (TTC) case [8]. We consider the problem of modularizing a system design, given as a class model, so that coherence and coupling are optimized. Class models contain a set of *packages*, where each package includes a set of *classes*. Classes are related via *associations*. The following modifications are allowed: classes may be moved between packages, and empty packages may be deleted and created. Classes and associations may not be deleted or created. Optimality of models is measured using a metric called *HCLC*, indicating high cohesion and low coupling, based on the intra- and inter-package associations of the contained packages:

$$HCLC(\underline{M}) = \sum_{P \in \underline{M}} \frac{\#intraPackageAs(P)}{\#classes(P)^2} - \sum_{P_1, P_2 \in \underline{M}} \frac{\#interPackageAs(P_1, P_2)}{\#classes(P_1) * \#classes(P_2)}$$

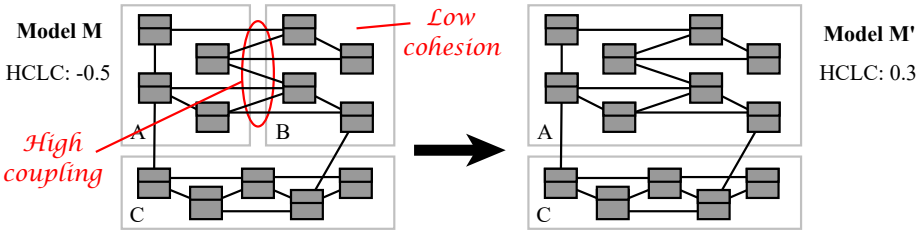


Fig. 1: Solution candidates: class models $M$ and $M'$.

Consider an instance of this problem where models $M$ and $M'$ in Fig. 1 are solution candidates. Packages are denoted using light-gray boxes; classes and associations are denoted using filled squares with black lines between them. The HCLC score of $M$ is -0.5: Packages $A$ and $B$ have few intra-package associations (low cohesion), and numerous inter-package associations (high coupling). $M'$ improves this situation: after merging packages $A$ and $B$, their inter-package associations have become intra-package ones, reflected in a HCLC score of 0.3.

To identify improved solution candidates such as $M'$ automatically, we can use a genetic algorithm, employing the HCLC metric as the fitness function. Using class models directly as solution candidates, we need to provide mutation and crossover operators. In the following, we focus on the mutation operator, which can be specified as an in-place model transformation. Fig. 2 shows three possible candidate rules for this purpose, specified using Henshin [9]. In these rules, nodes represent model elements, edges represent links, and all elements have a type and one of the actions «create», «delete», «preserve», and «forbid». Rule `moveClass` moves a class between two packages by deleting its containment edge and creating a new one. Rule `deletePackage` deletes a package from the model. The «forbid» part in this rule ensures that the package to be deleted is empty, which is necessary to comply with the constraint that classes may not be deleted. Rule `moveFourClasses` moves *four* classes between particular packages.

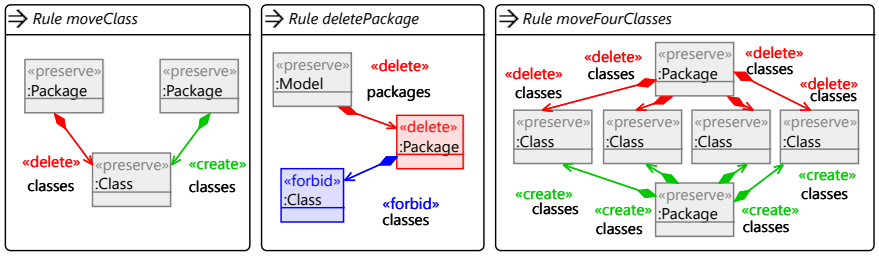

Fig. 2: Henshin rules used to define a mutation operator.

To pinpoint the drawback of the existing minimal mutation operators, assume a mutation operator based on the rules `moveClass`, `deletePackage` and a third rule `createPackage` for creating packages, based on the rationale that these rules have all the ingredients to enumerate all possible solution candidates. Indeed, model $M'$ can be obtained from $M$ by applying `moveClass` repeatedly to move all classes in $B$ to $A$, and deleting $B$ via `deletePackage`. However, in a genetic algorithm, the bottleneck of this process is getting the order of rule applications right: as we apply `moveClass` at randomly selected places, we often move classes from $A$ and $C$ to $B$, including classes that we earlier moved away from $B$. We may eventually discover $M'$, but the road is long and paved with meritless candidates.

An alternative mutation operator can be defined by adding `moveFourClasses` to the rule set. This additional rule gives us a shortcut that allows us to identify $M'$ at a minimum of two rule applications instead of five. Therefore, it is clearly tempting to assume that this mutation strategy is more efficient. However, while the manual effort to define this particular mutation operator is low, we only have an intuition, but no evidence that this operator is more efficient in the overall problem class. Worse, there is no guarantee that we did not miss yet another, even more efficient mutation operator. A systematic strategy to design efficient mutation operators for a particular problem class is generally lacking.

To address these issues, in this paper, we introduce *FitnessStudio*, a technique for generating efficient, problem-tailored mutation operators automatically. Specifically, we make the following contributions:

- A **two-tier framework** of nested genetic algorithms: In the lower tier, we consider a regular search over concrete solution candidates. In the upper tier, we "train" the mutation operator of the lower tier (Sec. 2).
- A **higher-order transformation** for mutating the rules of the lower-tier mutation operator, focusing on the goal to optimize its efficiency (Sec. 3).
- An **implementation** using the Henshin transformation language (Sec. 4).
- A **preliminary evaluation** based on a benchmark scenario. Using the generated mutation operator, the result quality was significantly improved compared to previous solutions, without sacrificing performance (Sec. 5).

The key idea to train the mutation operator on concrete examples is inspired by the notion of *meta-learning* [10], that refers to the tuning of mutation operations in the context of genetic programming. Our technique is the first to generate mutation operators for search-based MDE. We discuss the specific requirements of this use-case in Sect. 3, and survey related work in Sec. 6.

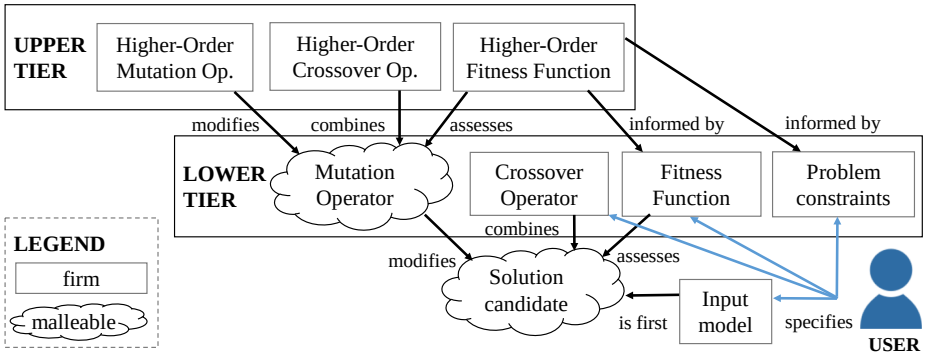


Fig. 3: Overview.

## 2 Framework overview

Given a search problem in which solution candidates are expressed as models, the goal of our technique is to generate an efficient mutation operator. To this end, we introduce a two-tier framework of nested genetic algorithms. As illustrated in Fig. 3, the upper tier optimizes the mutation operator of the lower tier. An interesting feature is that the mutation operator of the lower tier is *malleable*, in the same sense that the solution candidates are. In other words, the upper tier allows us to consider a *family* of genetic algorithms in the lower tier.

- As **solution candidates**, we consider models directly, i.e., we do not use a dedicated encoding. This setting allows us to modify the solution candidates using transformation rules that can be modified using higher-order rules.
- The **lower tier** of our framework includes several components that are customized by the user to the problem at hand: the fitness function, the crossover operator, and problem constraints (e.g., in our example, the constraint that classes may not be deleted). This manual customization effort is the same as in state-of-the-art search-based MDE approaches. In contrast to these approaches, the mutation operator is generated fully automatically. The mutation operator comprises a set of transformation rules that are orchestrated by a simple algorithm as described later.
- The **upper tier** is generic in the sense that it remains constant over arbitrary problems. Its most sophisticated feature is the mutation operator that uses a higher-order transformation to modify the lower-tier mutation operator. The crossover operator combines two randomly selected subsets of the rule sets of the two considered mutation operators. The higher-order fitness function assesses fitness in terms of the fitness of the fittest solution candidate produced in a sample run on a given input model. More sophisticated fitness functions as supported by multi-objective genetic algorithms are generally possible in this framework, although we did not explore this option in our implementation. Constraints imposed on the lower-tier mutation operator are checked directly after upper-tier mutations.

## 3 Upper-Tier and Lower-Tier Mutation Operators

This section is dedicated to our main technical contribution, the upper- and lower-tier mutation operators. We revisit necessary preliminaries, fix requirements based on our particular goals, and show the rules and their orchestration included in both levels.

### 3.1 Preliminaries

First, we revisit Henshin's meta-model, which defines an abstract syntax of rules. This meta-model acts in two ways as the foundation of our technique, as we transform rules using higher-order transformation rules. For clarity, we use the term **domain rules** for rules that are applied to domain models (in the example, class models), and **HOT rules** for rules that are applied to domain rules.

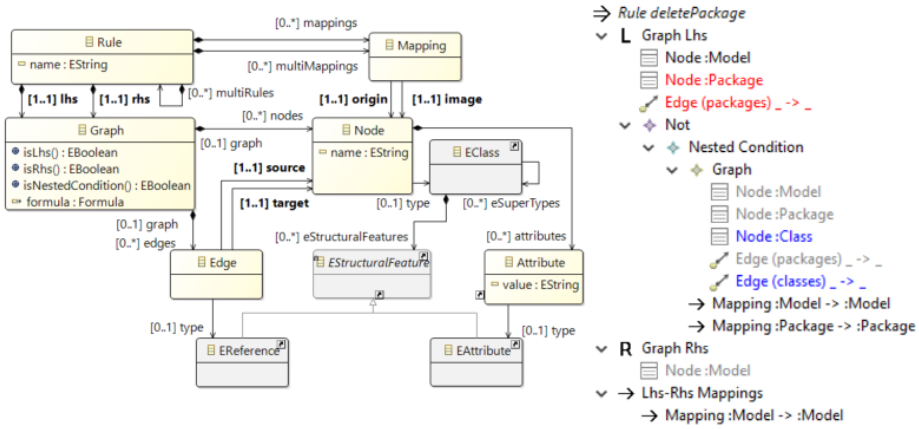


Fig. 4: Henshin meta-model and example domain rule.

Fig. 4 shows the relevant excerpt of the meta-model, and an example domain rule `deletePackage` created using the meta-model. The domain rule is the same one as shown in Fig. 2, but, to illustrate the meta-model, shown in a tree-based concrete syntax, rather than the more concise graphical concrete syntax.

A *rule* has a left-hand and a right-hand side *graph*, in short, LHS and RHS. A graph contains a set of *nodes* and a set of *edges*, so that each edge runs between a pair of nodes. Nodes can have *attributes*. Nodes from different graphs can be declared as identical using *mappings*. A mapping between an LHS and a RHS graph specifies that the respective node is a «preserve» node, i.e., it will be not be affected by the application of the rule. Nodes, edges, and attributes have types, which are references to suitable EClasses, EReferences, and EAttributes from the domain model's meta-model. The example rule includes as types the EClasses `Model`, `Package`, `Class`, and the EReferences `packages` and `classes`.

In the example rule, the «delete» and «preserve» elements from the graphical syntax correspond to distinct ways of including these elements in the containment tree: The `Package` node and its incoming containment edge are to be

deleted by the application of the rule and, thus, only appear in the LHS. The `Model` node is to be preserved by applications of the rule, and therefore appears in the LHS and the RHS, with a dedicated mapping to specify identity.

A further concept of rules are negative application conditions (NACs), which can be nested using a Boolean formula. For simplicity, we omit a large portion of the meta-model describing NACs.[1] At this time, it's sufficient to know that each NAC specifies a distinct graph, and its meaning is to «forbid» the existence of this graph in the input model. In the example rule, to represent the connection to the LHS, this graph contains some *context nodes and edges* to represent elements from the LHS, as specified via mappings. The forbidden part, highlighted in blue, includes all non-context-elements, here, the `Class` node and `classes` edge.

Rules can contain further rules, which are then referred to as *multi-rules*. During the application of a rule with multi-rules, the actual rule or *kernel rule* is applied first, and afterwards each multi-rule is applied as often as possible to the same matching site. Mappings running between a multi-rule and its containing rule are called *multi-mappings*.

### 3.2 Requirements

As a basis for design decisions influencing our higher-order transformation, we stipulate the following requirements.

**R1: Plausibility.** It is tempting to design a completely unconstrained rule mutation mechanism that can produce all possible domain rules. Yet, a large class of domain rules is actually undesirable and can be avoided for better efficiency: *disconnected domain rules*, where the graphs in the rule are composed of multiple connected components of nodes and edges, may lead to the creation and deletion of elements outside of the containment tree. Therefore, domain rules shall be altered in such a way that the result rule represents a connected graph.

**R2: Size variation.** Generally, it cannot be determined in advance how many rules an efficient mutation operator will consist of. Therefore, in addition to mutating the involved rules, the higher-order mutation operator needs to ensure that different sizes of the mutation rule set are explored.

**R3: Validity.** Problem constraints as specified by the user can be violated by specific domain rules, e.g., in the running example, a rule that deletes classes. To avoid such offending rules without losing the generality of the HOT, we can discard them directly after a mutation. Moreover, the output models of the framework must be valid concerning well-formedness and problem constraints. To ensure that our mutation operators contribute to this goal, we check the validity of the produced models during the upper-tier fitness evaluation.

**R4: Performance.** The mutation operators on both meta-levels involve a step where a rule is matched to an input model, an NP-hard task [11] that is applied many times during the identification of solution candidates. Since we generally only need to identify a single match, rather than all possible ones, many rules are simple enough to avoid notable performance drawbacks. To deal with occasional problematic ones, a timeout can be applied.

[1] Details are found at https://wiki.eclipse.org/Henshin_Transformation_Meta-Model

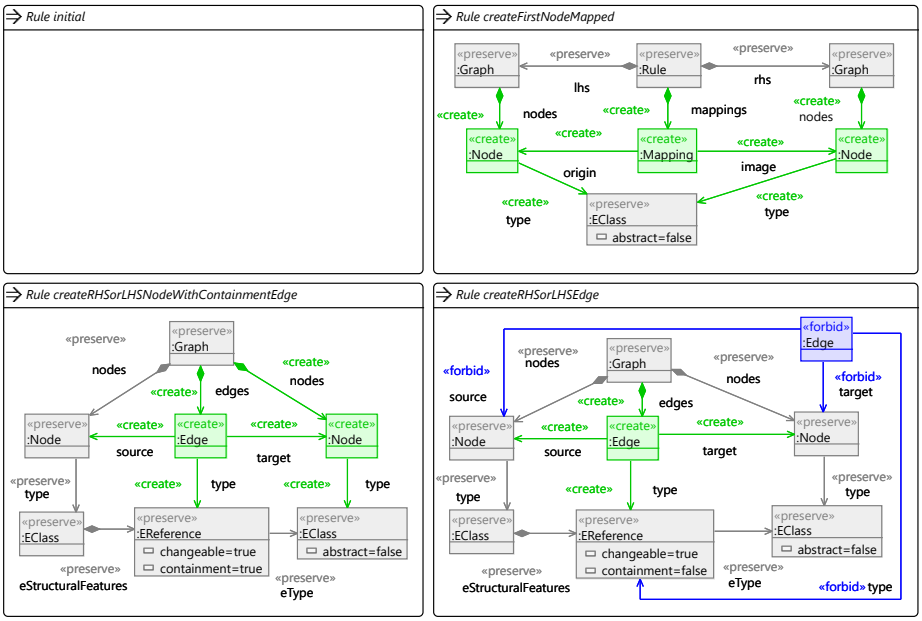


Fig. 5: Empty domain rule *initial* and three selected higher-order-transformation rules for creating basic rule elements.

### 3.3 Upper-tier mutation operator

Our upper-tier mutation operator includes a set of HOT rules and a simple algorithm to orchestrate their application. We first walk through the rules, starting with simple HOT rules addressing basic features of domain rules, and then move to HOT rules for advanced rule features and, finally, rule orchestration.

The initial input of our HOT is an empty domain rule, called `initial`. Fig. 5 shows this rule plus three selected HOT rules. Since our aim is to produce connected rules (**R1**), nodes are generally added together with an edge that relates them to the rest of the graph. The only exception is rule `createFirstNodeMapped`, which is only applied to the initial rule, so that it adds a node of the same type to the LHS and RHS. Both nodes are related via a mapping, yielding a «preserve» node in total. From there, we can extend the graph with additional nodes and edges. Rule `createRHSorLHSNodeWithContainmentEdge` adds a node with its incoming containment edge to the LHS or the RHS, i.e., a «delete» or «create» node. Modeling the containment edge is important, since creating or deleting a model element requires the addition or removal of this element in the containment tree. Rule `createRHSorLHSEdge` adds further «delete» or «create» edges; the «forbid» part avoids the creation of multiple edges of the same type between the same nodes, which is illegal. Variants of these rules exist for the addition and removal of «preserve» nodes and edges. In all rules, types of newly created nodes and edges are chosen arbitrarily based on the provided meta-model.

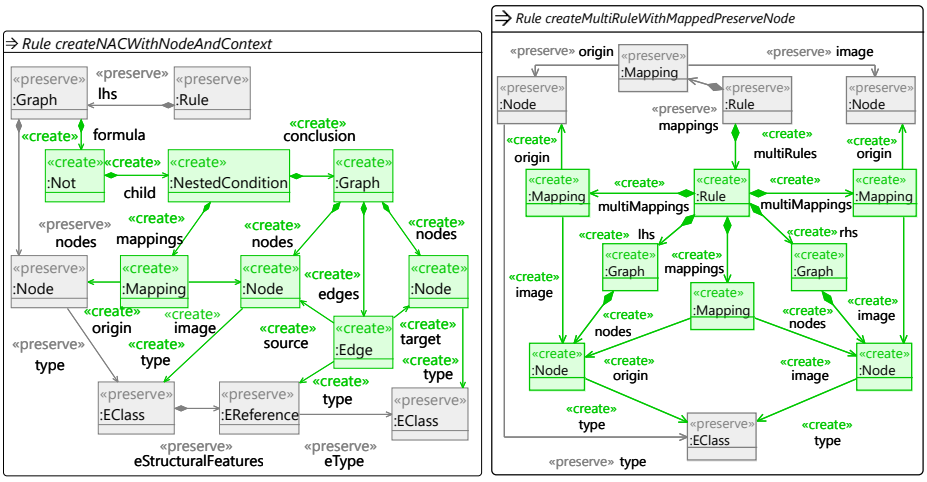


Fig. 6: Higher-order mutation rules for creating advanced rule elements.

HOT rules for introducing the advanced concepts of NACs and multi-rules are shown in Fig. 6. Based on the requirement to create connected rules, the goal of HOT rule `createNACWithNodeAndContext` is to create a NAC whose elements are related to elements of the containing rule, which reflects the notion of *context elements* as explained in Sect. 3.1. The HOT rule assumes a domain rule where the LHS contains a node, so that a reference between the node's type `EClass` and some other `EClass` exists, allowing us create an edge between these nodes. Given such a rule, the NAC is added by creating a negated application condition (called *NestedCondition*, since Henshin allows nesting them) to the LHS. The conclusion of the NAC is a graph which is populated with two nodes and an edge relating the nodes. One of the nodes is mapped to the existing LHS node, and thus becomes the context node. The non-context node and the edge are «forbid» elements. For completeness, a dual opposite of this rule exists, where the direction of the edge running between the nodes is inverted.

Similarly, rule `createMultiRuleWithMappedPreserveNode` aims to create a multi-rule which contains a context part from the original rule. We use a «preserve» node from the original rule as context, i.e., a pair of LHS and RHS nodes with a mapping. The multi-rule is created so that its LHS and RHS contained a «preserve» node as well, which is mapped to the original rule using multi-mappings. The multi-rules and NACs created using these rules can be further populated using the simple HOT rules as introduced above.

The algorithm for orchestrating the HOT rules is introduced in Fig. 7. Lines 1–9 address the requirement to achieve variability in the size of the rule set: we initially start with a fixed number of domain rules (stored in a constant `INIT`, not shown). However, since a more efficient mutation operator might require additional or fewer rules, we change the size of the set dynamically by removing or rules in case a certain probability threshold, `DIE` or `REPLICATE`, is reached (**R2**). In lines 10–21, we iterate over pairs of domain rules and HOT rules, applying the

```
1 // Achieve variability in the rule set size
2 for (Rule rule : domainRules) {
3    double fate = Math.random();
4    if (fate < DIE && domainRules.size() > 2) {
5       domainRules.remove(rule);
6    } else if (fate > REPLICATE) {
7       domainRules.add(domainRule.createCopy());
8    }
9 }
10 // Randomly apply HOT rules to domain rules
11 for (Rule rule : domainRules) {
12    for (Rule hotRule : hotRules) {
13       if (Math.random() > MUTATE_HOT) {
14          EGraph graph = new EGraphImpl(rule);
15          RuleApplication app = new
                RuleApplicationImpl(graph, hotRule);
16          boolean applied = app.execute(null);
17          if (applied && violatesConstraint(domainRule))
18             app.undo(null);
19       }
20    }
21 }
```

Fig. 7: Orchestration of higher-order mutation operator.

latter to the former in case another probability threshold `MUTATE_HOT` is met. To perform the mutation, we wrap the domain rule into an `EGraph`, which encapsulates the input model in a Henshin transformation for a `RuleApplication` of the HOT rule. Constraints imposed on domain rules (e.g., in our running example, classes may not be deleted or created) are checked in line 19, leading to an undo of the mutation if a constraint is violated (**R3**). Specific mutated rules might be affected by performance bad smells [12] that will not become obvious until the higher-order fitness evaluation. To keep these inefficient rules from spoiling the overall performance, we support a user-specified timeout duration that is applied to the complete evolution iteration (**R4**).

### 3.4 Lower-tier mutation operator

The lower-tier mutation operator in our framework consists of the domain rules generated by the upper tier, and a short algorithm for orchestrating these rules. The algorithm, shown in Fig. 8, copies the current solution candidate to mutate it. The mutation rules to be applied are selected using a static threshold, `MUTATE_DOM`, to specify the mutation probability.

```
1 EObject mutated =
     EcoreUtil.copy(mutated);
2 EGraph graph = new
     EGraphImpl(mutated);
3 for (Rule rule : domainRules) {
4    if (Math.random() > MUTATE_DOM)
5       new RuleApplicationImpl(graph,
           rule).execute(null);
6 }
```

Fig. 8: Orchestration of domain rules.

## 4 Implementation

We implemented FitnessStudio based on Henshin and a genetic algorithm, providing the implementation at `https://github.com/dstrueber/fitnessstudio`. The genetic algorithm, available at `https://github.com/lagodiuk/genetic-algorithm`, is a simple single-objective one. The rationale for choosing this algorithm was its convenient application to our technique, and positive experience made with a solution to the TTC 2016 case [13].

As illustrated in Fig. 9, in each iteration, the algorithm takes each chromosome of the parent population, mutates it, and performs a crossover with a randomly chosen chromosome from the parent population. The results are merged with the parent population and trimmed to obtain the same number of chromosomes again.

```
for (Chromosome c1 :
    parentPopulation) {
   Chromosome c2 = c1.mutate();
   Chromosome c3 =
       parentPopulation.rand();
   population.addAll(c2.crossover(c3));
}
population.addAll(parentPopulation);
population.sortByFitness();
population.trim(populationSize);
```

Fig. 9: Evolution iteration in the used algorithm.

We assigned the thresholds introduced in Sects. 3.3 and 3.4 as follows, based on the rationale to have a "balanced" amount of mutation: *{INIT=4, DIE=0.33, REPLICATE=0.8, MUTATION_HOT=0.8, MUTATION_DOM=0.4}*. Further experimentation with dynamic thresholds may lead to improved results.

## 5 Preliminary Evaluation

To evaluate the efficiency of the generated mutation operators, we investigated two research questions: **(RQ1)** What is the quality of the solutions produced by the generated mutation operators? **(RQ2)** How do the generated mutation operators affect performance? We evaluated these questions using the "Class Responsibility Assignment" (CRA) case of the Transformation Tool Contest 2016 [8], a case that qualifies as a first benchmark for search-based model driven engineering, in which ten existing solutions are available.

**Scenario.** In the CRA case, the task is to create a high-quality decomposition of a given class model. The input model comprises a set of *features*, i.e., methods and attributes with functional and data dependencies between them. Features can be assigned arbitrarily to classes, so that a combined coherence and coupling score, called *CRA index*, is maximized.

| Models | **A** | **B** | **C** | **D** | **E** |
|---|---|---|---|---|---|
| Attributes | 5 | 10 | 20 | 40 | 80 |
| Methods | 4 | 8 | 15 | 40 | 80 |
| Data dep. | 8 | 15 | 50 | 150 | 300 |
| Function dep. | 6 | 15 | 50 | 150 | 300 |

Table 1: Input models (from [8]).

In our evaluation, we used five input models of varying size that were provided with the case description, since they allow a direct comparison of our technique

and the existing solutions. Detailed information on these models is provided in Table 1. To discuss the efficiency of our technique, we consider two baselines: the best-performing and a median solution the TTC case study, allowing a detailed comparison to the state of the art. The best-performing solution was based on the Viatra-DSE framework [5]. For our comparison, we used the values reported in the final proceedings version [14], which were obtained on a system comparable to the one used in this paper. The information about median solutions – we considered median scores and times separately for all input models – was obtained from a publicly available overview spreadsheet[2]. Where available, we used the scores of the improved post-workshop versions. In this table, it is not specified if the best or average solution is reported, and the execution times were measured on different systems. Both aspects threaten the accuracy of comparisons based on these data, but arguably not to a disqualifying extent (as discussed later).

**Set-up.** Our technique has an initial cost for deriving the mutation operator. To mitigate this cost, we aimed to ensure that the same mutation operator can be reused over problems in the same problem class, i.e., is not overfitted. To this end, we worked with training and evaluation sets, so that the actual benchmark was the evaluation set. As the training set, we used model C, since it enabled a compromise between representative size and training duration. In the provided input model, we encapsulated each given function and attribute in a singleton class. The crossover operator was based on combining randomly selected classes from both considered models. We evaluated quality in terms of the CRA index of the produced models, as well as performance in terms of execution time. All rules, models and data produced during our experiments are available at `https://github.com/dstrueber/fitnessstudio`.

*Preparation: generation of the mutation operator.* We applied FitnessStudio to input model C, configuring the upper tier to a population size of 40 and 20 iterations, and the lower tier to population size 2 and 40 iterations. We repeated the generation 10 times; the median run took 8:50 minutes. The best-performing mutation operator produced by FitnessStudio contained nine rules that each reassigned between one and three features. Interestingly, all rules shared a common pattern, where the specified target class of a feature $F$ contains another feature $F'$ so that a dependency between $F$ and $F'$ exists, thus contributing to cohesion. Each rule had one NAC; no rule had a multi-rule.

*Benchmark measurement.* We applied the genetic algorithm together with the top-scoring generated mutation operator to models A–E. To study the variability of the produced results, we repeated the experiment 30 times, using a population size of 40 and 400 iterations. All experiments were performed on a Windows 10 system (Intel Core i7-5600U, 2.6 GHz; 8 GB of RAM, Java 1.8 with 2 GB maximum memory size).

**Results.** Table 2 and Fig. 10 show the results of our experiments.

**RQ1: Result quality.** In all cases, fitness of our best identified solution candidates, as measured in terms of CRA, was equal to (A–B) or greater than (C–

[2] For the original spreadsheet, see http://tinyurl.com/z75n7fc – for the computation of medians, see our spreadsheet at https://git.io/vyGpJ

| Input model | *Median TTC solution* | | *Viatra-DSE* | | | *FitnessStudio* | | |
|---|---|---|---|---|---|---|---|---|
| | CRA | Time | CRA best | CRA median | Time median | CRA best | CRA median | Time median |
| A | 3.0 | 00:02.284 | 3.0 | 3.0 | 00:04.729 | **3.0** | **3.0** | **00:05.260** |
| B | 3.3 | 00:04.151 | 4.0 | 3.8 | 00:13.891 | **4.0** | **3.2** | **00:11.511** |
| C | 1.8 | 00:24.407 | 3.0 | 2.0 | 00:17.707 | **4.0** | **3.3** | **00:24.837** |
| D | 0.4 | 01:42.685 | 5.1 | 2.9 | 01:19.136 | **8.1** | **6.5** | **01:30.799** |
| E | 0.2 | 11:04.802 | 8.1 | 5.0 | 09:14.769 | **17.2** | **12.8** | **05:09.194** |

Table 2: Results, times being denoted in terms of mm:ss.xxx.

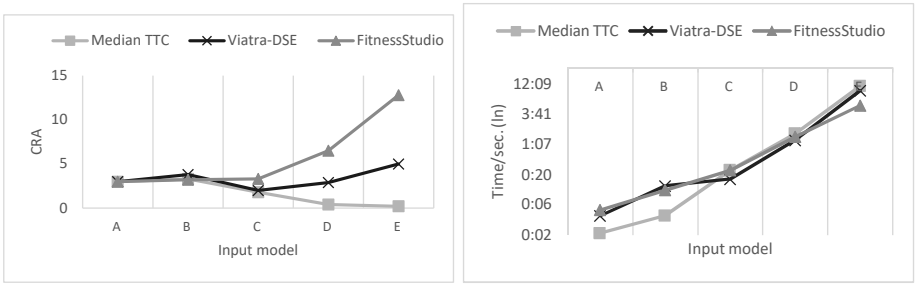


Fig. 10: Median CRA scores and execution times.

E) both baseline solutions. Strikingly, the delta between the best found solution of Viatra-DSE and FitnessStudio increased with the size of the model, ranging from no improvement in the case of A and B, to a strong improvement in the case of E. The median CRA improved in a similar manner, except for model B, where our mutation operator was too coarse-grained to yield the optimal solution as consistently as ViatraDSE. Altogether, our observations indicate an improved quality in particular for larger models.

**RQ2: Performance.** The time required to produce solutions using the mutation operator from FitnessStudio was in the same order of magnitude as Viatra-DSE. For the largest considered model E, we actually achieved a speed-up by factor 1.8, although a general trend towards better scalability is not obvious.

The results demonstrate that the mutation operators produced by our technique can produce better solution candidates than the state of the art, without sacrificing performance. We still need to evaluate our technique in a larger benchmark set of cases to confirm the generality of this conclusion.

**Limitations and Threats to Validity.** A limitation is that the mutation operator generation adds an initial performance overhead to applications of our technique. The severity of this limitation depends on the use-case: the overhead is less important if the search is to be ran repeatedly, either on different instances of the problem or changed versions of the same instance. As a lesson learned

from our experiments, generation time can be reduced by training the mutation operator on a representative, but not overly large model.

External validity of our results is threatened by use of a single benchmark case. Particularly, the low complexity of the involved meta-model is a caveat against generalization to more complicated cases. In fact, we might need additional HOT rules to support meta-models where generalization plays a more important rule, to capture different combinations of relating `EClass`es (and supertypes thereof) via `EReference`s. To avoid pattern duplication during the creation of these variants, a promising strategy is to express them using variability-based rules [15,16]. Reliability of our comparison is threatened by the variability of baseline data for the median TTC solution. Still, even if we assume a conservative interpretation where all values are medians, these scores would still be below the median scores of the Viatra-DSE and the FitnessStudio solutions.

## 6 Related Work

**Combinations of SBSE and MDE** can be distinguished in terms of whether they operate on the input model directly [7], or encode it to a custom representation [5,6]. In addition, some approaches optimize the input model, whereas others optimize the orchestration of rules applied to the model. In all cases that we are aware of, including the solutions submitted to the TTC case [8], the mutation operators for the considered problem are defined manually. In fact, the Henshin solution [13] experimented with a selection of different manually defined mutation rules. The authors observed a drastic impact of the chosen mutation strategy, an observation that inspired the present work. Remarkably, those manually defined mutation operators are outperformed by our generated one. Finally, Mkaouer and Kessentini [17] haved used a genetic algorithm for optimizing transformation rules; yet, the baseline scenario of this approach is a regular transformation, rather than a genetic operator in an optimization process.

**Rule generation** for arbitrary meta-models is supported by an approach of Kehrer at el. [18] for the generation of consistency-preserving editing rules. A benefit of this approach is that the produced rule set is complete in the sense that it can produce all valid models. However, our example in Sect. 1 illustrates why such a rule set may not be most effective for our specialized usecase. Since rules are a particular type of model, their generation can also be enabled by using a model generator. In this context, Popoola et al. [19] provide a DSL for specifying a model generation strategy. In contrast, our approach aims to discover efficient mutation operators automatically, without relying on user input.

**Model mutation** is also an important concept in other contexts than SBSE. Focusing on testing, Troya et al. [20] and Alhwikem et al. [21] have proposed approaches for the systematic design of mutation operators for ATL and arbitrary domain-specific languages, respectively. Both approaches involve the definition of generic or abstract mutation operators that are instantiated to obtain concrete ones. In addition, the approach by Troya et al. uses an ATL higher-order transformation to manipulate the generated operators. Similarly to Kehrer's approach, the generation strategies in these works are complementary to ours, since they

aim at completeness, while ours aims at fitness, leading to a different set of requirements. Wodel [22] is an expressive domain-specific language for model mutation, providing high-level mutation primitives so that users can specify purpose-specific mutation strategies. In contrast, our higher-order-transformation does not rely on a user specification of the mutation strategy, as it aims to discover efficient mutation operators automatically.

**Mutation operator improvement** has attracted the attention of the genetic programming community. A group of approaches based on *self-adaptive mutation* [23] modifies the probability of performing a mutation over the course of the genetic algorithm's application, rather than the mutation structure itself. Philosophically most similar to ours is the work by Woodward and Swan [10], which is based on the notion of meta-learning, and modifies mutation operators encoded in general-purpose-language using a notion of register machines. The work by Martin and Tauritz [24] expands on that idea as it aims to optimize the algorithm *structure*, rather than the involved genetic operators.

## 7 Conclusion and Future Work

The goal of this work is to provide a more systematic alternative to the current ad-hoc development style of mutation operators for search-based model-driven engineering. Our fundamental contribution is a higher-order transformation that generates and optimizes domain-specific mutation operators, as inspired by the notion of meta-learning. The early results obtained based on the TTC case indicate the generated mutation operators can be used to produce improved results compared to the state of the art, without sacrificing performance.

The present work opens up several exciting directions for future work. A fascinating problem concerns the self-applicability of our technique: Our higher-order mutation was developed in an ad-hoc fashion, not unlike the domain-specific mutation operators we aim to improve on. Can we obtain further improvements if we optimize the higher-order mutation as well, using a second-level higher-order-transformation, as would lead to *meta-meta-learning*? Moreover, the principles considered in this work could also inform the generation of cross-over operators. Finally, we aim to apply the approach to a broader variety of use-cases, including the refactoring of transformation rules [25], the prioritization of clones during quality assurance [26], and trade-off management in model-based privacy analysis [27]. In these scenarios, we also intend to study the impact of the used genetic algorithm, and the use of our technique for multi-objective optimization.

**Acknowledgement** This research was partially supported by the research project Visual Privacy Management in User Centric Open Environments (supported by the EU's Horizon 2020 programme, Proposal number: 653642).

## References

1. Harman, M., Jones, B.F.: Search-based software engineering. Information and software Technology **43**(14) (2001) 833–839
2. Fleck, M., Troya, J., Wimmer, M.: Search-based model transformations. Software: Evolution and Process **28** (2016) 1081–1117

3. Debreceni, C., Ráth, I., Varró, D., Carlos, X.D., Mendialdua, X., Trujillo, S.: Automated model merge by design space exploration. In: FASE, Springer (2016) 104–121
4. Fleck, M., Troya, J., Kessentini, M., Wimmer, M., Alkhazi, B.: Model transformation modularization as a many-objective optimization problem. IEEE Transactions on Software Engineering (2017)
5. Abdeen, H., Varró, D., Sahraoui, H., Nagy, A.S., Debreceni, C., Hegedüs, Á., Horváth, Á.: Multi-objective optimization in rule-based design space exploration. In: ASE, ACM (2014) 289–300
6. Fleck, M., Troya, J., Wimmer, M.: Search-Based Model Transformations with MOMoT. In: ICMT, Springer (2016) 79–87
7. Zschaler, S., Mandow, L.: Towards model-based optimisation: Using domain knowledge explicitly. In: MELO. (2016) 317–329
8. Fleck, M., Troya, J., Wimmer, M.: The class responsibility assignment case. In: TTC. (2016) 1–8
9. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: MoDELS. Springer (2010) 121–135
10. Woodward, J.R., Swan, J.: The automatic generation of mutation operators for genetic algorithms. In: GECCO. (2012) 67–74
11. Mehlhorn, K.: Graph Algorithms and NP-completeness. Springer-Verlag New York, Inc., New York, NY, USA (1984)
12. Tichy, M., Krause, C., Liebel, G.: Detecting Performance Bad Smells for Henshin Model Transformations. In: AMT. (2013)
13. Born, K., Schulz, S., Strüber, D., John, S.: Solving the Class Responsibility Assignment Case with Henshin and a Genetic Algorithm. In: TTC. (2016) 45–54
14. Nagy, A.S., Szárnyas, G.: Class Responsiblity Assignment Case: a Viatra-DSE Solution. In: TTC. (2016) 39–44
15. Strüber, D., Schulz, S.: A tool environment for managing families of model transformation rules. In: ICGT. Springer (2016)
16. Strüber, D.: Model-Driven Engineering in the Large: Refactoring Techniques for Models and Model Transformation Systems. PhD thesis, Philipps-Universität Marburg (2016)
17. Mkaouer, M.W., Kessentini, M.: Model transformation using multiobjective optimization. Advances in Computers **92** (2014) 161–202
18. Kehrer, T., Taentzer, G., Rindt, M., Kelter, U.: Automatically deriving the specification of model editing operations from meta-models. In: ICMT, Springer (2016) 173–188
19. Popoola, S., Kolovos, D.S., Rodriguez, H.H.: EMG: A Domain-Specific Transformation Language for Synthetic Model Generation. In: ICMT, Springer (2016) 36–51
20. Troya, J., Bergmayr, A., Burgueño, L., Wimmer, M.: Towards systematic mutations for and with ATL model transformations. In: Workshop on Mutation Analysis. (2015) 1–10
21. Alhwikem, F., Paige, R.F., Rose, L., Alexander, R.: A systematic approach for designing mutation operators for MDE languages. In: MoDEVVa. (2016) 54–59
22. Gómez-Abajo, P., Guerra, E., de Lara, J.: A domain-specific language for model mutation and its application to the automated generation of exercises. Computer Languages, Systems & Structures (2016)
23. Smullen, D., Gillett, J., Heron, J., Rahnamayan, S.: Genetic algorithm with self-adaptive mutation controlled by chromosome similarity. In: CEC, IEEE (2014) 504–511
24. Martin, M.A., Tauritz, D.R.: Evolving black-box search algorithms employing genetic programming. In: GECCO, companion volume, ACM (2013) 1497–1504
25. Strüber, D., Rubin, J., Arendt, T., Chechik, M., Taentzer, G., Plöger, J.: RuleMerger: Automatic Construction of Variability-Based Model Transformation Rules. In: FASE, Springer (2016) 122–140
26. Strüber, D., Plöger, J., Acreţoaie, V.: Clone detection for graph-based model transformation languages. In: ICMT, Springer (2016) 191–206
27. Ahmadian, A.S., Strüber, D., Riediger, V., Jürjens, J.: Model-based privacy analysis in industrial ecosystems. In: ECMFA, Springer (2017) accepted.