BACHELOR THESIS
COMPUTING SCIENCE

RADBOUD UNIVERSITY

Generating Mutation Operators for a Search-Based Model-Driven Implementation of the Next Release Problem

Author:
Niels van Harten (s1012159)
`niels@vharten.com`

First supervisor/assessor:
Dr D.G.F. Strüber (Daniel)
`d.strueber@cs.ru.nl`

Second assessor:
Dr N.H. Jansen (Nils)
`n.jansen@cs.ru.nl`

June 13, 2021

Abstract

Any company developing and maintaining software products faces the next release problem (NRP). The NRP seeks to find the optimal subset of tasks to include in the next release of a companies product, to minimize development cost and maximize customer satisfaction. The NRP is an NP-hard, multi-objective software engineering problem that we solve using search-based software engineering (SBSE) techniques in an intuitive new way.

Within the field of SBSE, genetic algorithms (GA's), mimicking the biological process of evolution, are often used to solve software engineering problems. A GA improves an initial population of candidate solutions over time using manipulation operators favoring reproduction of good over bad solutions. However, it can be hard to come up with a representation for the candidate solutions as well as an efficient mutation operator.

Therefore, we combine SBSE with model-driven engineering (MDE) to represent candidate solutions as model instances and automatically generate mutation operators using a notion of meta-learning. We extend an available automated technique for generating efficient mutation operators to address the peculiarities of the NRP and use it to generate mutation operators. We subsequently combine these mutation operators with an available manual mutation operator for the NRP and compare result quality and performance to using only the manual mutation operator for both a single- and multi-objective implementation of the NRP. In most cases our results when combining the two are promising for both implementations showing improved result quality as well as performance.

Contents

Chapter 1

Introduction

What is the optimal subset of tasks to include in the next release of a companies product, to minimize development cost and maximize customer satisfaction? This is the next release problem (NRP, [5]), an NP-hard, multi-objective, software engineering problem that we will solve using search-based software engineering (SBSE, [17]) techniques in an intuitive new way.

SBSE is a subfield of software engineering and seeks to reformulate problems like the NRP as search problems consisting of three ingredients:

1) A problem representation

2) A fitness function to optimize towards

3) A set of manipulation operators

SBSE subsequently seeks to solve those reformulated problems using meta-heuristic techniques, in our case genetic algorithms. Genetic algorithms (GA's, [33]) mimic the biological process of evolution. They improve an initial population of candidate solutions over time using manipulation operators favoring reproduction of good over bad solutions. We use meta-heuristics because the NRP is NP-hard and therefore cannot be solved using exact optimization techniques for large scale problem instances [35].

To obtain the first ingredient of the reformulated problem, the problem representation, we can combine SBSE with model-driven engineering (MDE, [22]), which focuses on the use of precise models to support the development process. MDE solutions - models - can function as problem representation on which SBSE techniques can be applied directly. An advantage of using MDE is that we do not need to create a problem representation from scratch. In previous work, it has been shown that such a model-based problem encoding can lead to solutions with good performance behavior [21].

Given the representation, we want to optimize an initial population towards a fitness function, the second ingredient. We do this using the set of manipulation operators, where different search techniques require different operators. In our case, we use a genetic algorithm for which we first generate children from the population with a crossover operator. Next, we mutate those children using a mutation operator. Lastly, we evaluate the generated children resulting from our genetic algorithm by their customer satisfaction and cost by giving them a fitness score for both objectives.

However, it can be hard to come up with an efficient mutation operator. The developer has to rely on intuition, which may or may not be correct. Even when correct, one likely misses ideas for better solutions. Therefore we use FitnessStudio [29], a technique for generating efficient problem-tailored mutation operators automatically based on a two-tier framework. An upper tier optimization process tunes the mutation operator of the lower tier. The framework has been implemented using a mono-objective genetic algorithm and evaluated in a benchmark case of program refactoring where it showed promising results. This research extends that work by implementing the framework using a multi-objective genetic algorithm. Many software engineering problems, including the NRP, are in fact complicated multi-objective problems, which cannot be efficiently solved using single-objective solutions [19].

We will solve the next release problem intuitively by using a model as representation and combine generated mutation operators using FitnessStudio with a manually defined mutation operator and compare it to solving the problem using only the manually defined mutation operator. In particular, our contributions are the following:

- A **single-objective implementation of the NRP** in which we make two alterations to FitnessStudio and give a preliminary evaluation. We present an evaluation showing that combining an available mutation operator with our generated mutation operators significantly improves result quality for large models and when using *random* initialization performance is improved at the same time (Ch. 3).

- A **multi-objective implementation of the NRP** with a preliminary evaluation showing that combining an available mutation operator with our generated mutation operators in most cases significantly improves performance while also improving result quality for the multi-objective NRP (Ch. 4).

We provide the preliminaries needed for interpreting our results in Ch. 2. In Ch. 5 we survey related work.

Chapter 2

Preliminaries

2.1 The next release problem

Any company developing and maintaining software products sold to a range of diverse customers faces the next release problem (NRP, [5]). The NRP is about determining what should be included in the next release of a company's products. The company is faced with customer demands for a wide range of software enhancements where some enhancements will require (one or more) prerequisite enhancements. Besides, some customers are more valuable to the company than others so that the requirements of favoured customers will be viewed as having more importance than those of less favoured customers. At the same time, the different requirements will take widely differing amounts of time and effort to meet.

The challenge for the company is to select a set of requirements that is deliverable within their own budget and which meets the demands of their (important) customers. Making an incorrect decision can prove a serious mistake. Good customers can be lost if their requirements are not met; the company itself may go over budget or even not deliver on time if costs are not controlled.

2.2 Model representation for the NRP

Previous work found that MDE solutions can function as problem representation for the NRP directly, addressing the first key ingredient of SBSE [21]. A metamodel for this problem exists, provided as part of the MDEOptimiser project [9] and created using the Eclipse Modeling Framework (EMF, [28]). The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model [1].

The MDEOptimiser metamodel for the NRP is shown in Fig 2.1. A model instance consists of a number of *solutions* where a solution contains a subset of the $availableArtifacts$ called the $selectedArtifacts$. A software artifact

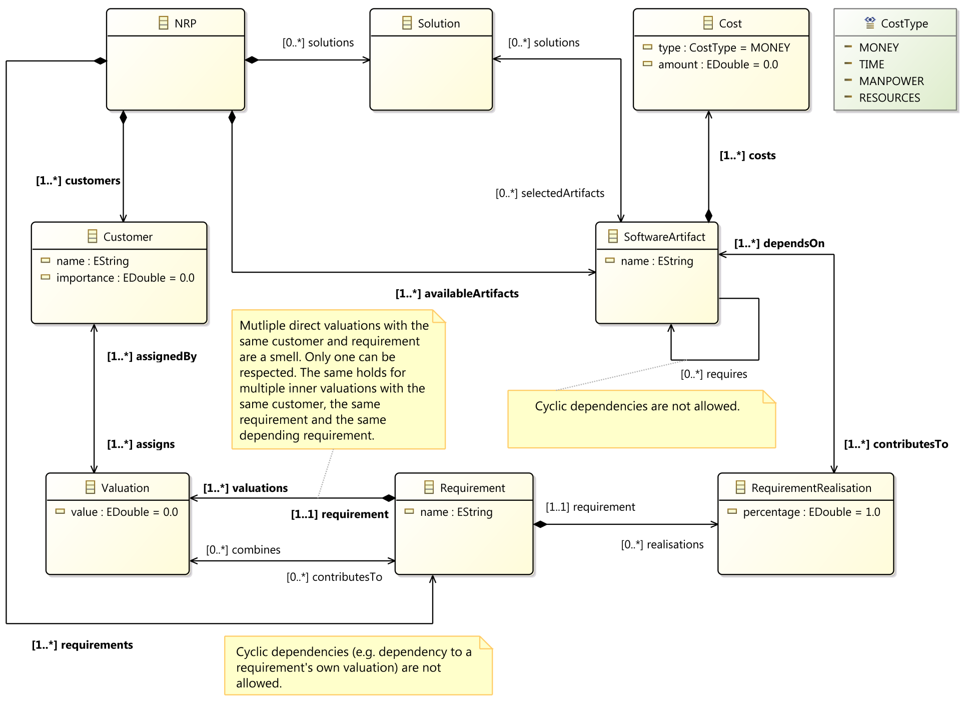


Figure 2.1: Metamodel of the NRP provided by MDEOptimiser.

has a *cost*, $contributesTo$ a $RequirementRealisation$ and it can be that an artifact $requires$ other artifacts (i.e. has dependencies). A requirement realisation $dependsOn$ one or more artifacts and can realise a $requirement$ that has one or more $valuations$ $assignedBy$ a $customer$.

The next release problem has two objectives. Given the above metamodel, the first is to minimize the total cost for the selected artifacts in a solution. The total cost is defined in the following way:

$$Cost = \sum_{sa \in SA'} cost(sa)$$

Here, SA' is the set of $selectedArtifacts$ and $cost(sa)$ is the cost for artifact sa. The second objective is to maximize customer satisfaction by realising requirements keeping in mind the value assigned to those requirements by customers and the importance of those customers.

$$Satisfaction = \sum_{c \in C} importance(c) \cdot satisfaction(c)$$

In this formula, C is the set of customers, $importance(c)$ is the importance of customer c and $satisfaction(c)$ is specified as:

$$satisfaction(c) = \frac{\sum_{v \in MDV(c)} value(v) \cdot fulfillment(requirement(v))}{\sum_{v \in MDV(c)} value(v)}$$

$MDV(c)$ is the set of all maximal valuations of direct requirements for customer c. Direct requirements are those which do not depend on other requirements. $value(v)$ is the value of valuation v and $fulfillment(requirement(v))$ calculates the highest degree to which the requirement of valuation v is fulfilled by either 1) direct realisations or 2) a combination of dependency requirements.

1) A requirement can be realised directly in one or more ways depending on the available $realisations$ for a given requirement. A $requirementRealisation$ is fulfilled if all $softwareArtifacts$ the realisation depends on are included in the solution as well as all their dependencies. If multiple realisations are fulfilled, the one with the highest percentage is chosen.

2) A requirement can also be fulfilled by a combination of other requirements. In that case, the level of fulfillment is determined by the weighted sum of the level of fulfillment of those dependencies.

Let us give the example from the MDEOptimiser implementation:
Requirement A depends on two Requirements B and C. The valuations connecting A to B and C have the values $v(B) = 2$, $v(C) = 4$. The level of fulfillments are $fulfillment(B) = 0.8$, $fulfillment(C) = 0.5$. The level of fulfillment for A then is $fulfillment(A) = (0.8 \cdot 2 + 0.5 \cdot 4)/6 = 0.6$. In case there is a direct realisation for A with a percentage of 0.8 the overall level of fulfillment will be $max(0.6, 0.8) = 0.8$.

If all $availableArtifacts$ are selected, the cost of the release as well as the customer satisfaction will be maximum. The goal is to find a good trade-off between maximizing customer satisfaction while minimizing the cost of all selected software artifacts.

2.3 Model transformation

Using the above model representation in EMF, we need a way to transform models while genetic algorithms rely on mutation and crossover of the solution candidates, in our case models. We will transform models in two ways.

First, using code in which the model elements can be seen as classes and links between elements as attributes of those classes. The relevant class structure can be automatically generated from the metamodel using EMF and then supports solution manipulation using methods calls in the following way:

$nrp.getSolutions().get(0).getSelectedArtifacts().add(artifact);$
$artifact.getSolutions().add(nrp.getSolutions().get(0));$

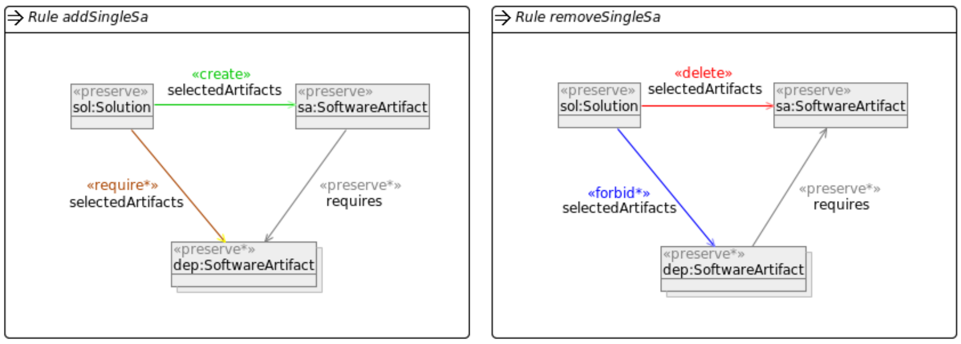


Figure 2.2: Henshin rules, part of the MDEOptimiser mutation operator.

The above code snippet first selects the first solution for nrp and adds $artifact$ to the selected artifacts. Then, it also adds a connection in the other direction. It adds the first solution of nrp to the solutions of $artifact$. We use this transformation method to apply crossover.

The second way to transform solution candidates is using a transformation language for EMF, called Henshin [4, 30]. Henshin is a rule-based model transformation language based on algebraic graph transformations and consisting of a tool set with editors and an interpreter engine. Using Henshin, we can specify rules to apply an in-place model transformation. These rules specify basic "match and apply" patterns. Fig. 2.2 shows two such mutation operator rules from the MDEOptimiser implementation. The model elements are represented by nodes and the links between model elements by edges. Every node and edge has one of the actions «create», «delete», «preserve», «require» or «forbid».

Rule *addSingleSa* adds *sa*, a software artifact, to *sol*, a solution. However, the action «requires» secures that a dependency of artifact *sa* is included in solution *sol*. Adding * to this action, «requires*», expands this rule to require that **all** dependencies of artifact *sa* are included in solution *sol*. Rule *removeSingleSa* removes artifact *sa* from solution *sol*. However, given the action «forbid*», it forbids that any of the dependencies of artifact *sa* are included in solution *sol*. We use this transformation method to apply mutation. Noteworthy is that although it is quite easy to come up with mutation rules that should improve the model quality at least in certain situations, it is hard to determine which set of mutation rules would be the most efficient.

2.4 FitnessStudio

FitnessStudio [29] is a two tier framework to generate efficient mutation operators for search-based model-driven engineering problems. An overview of this framework is given in Fig. 2.3. In this thesis, we extend the framework

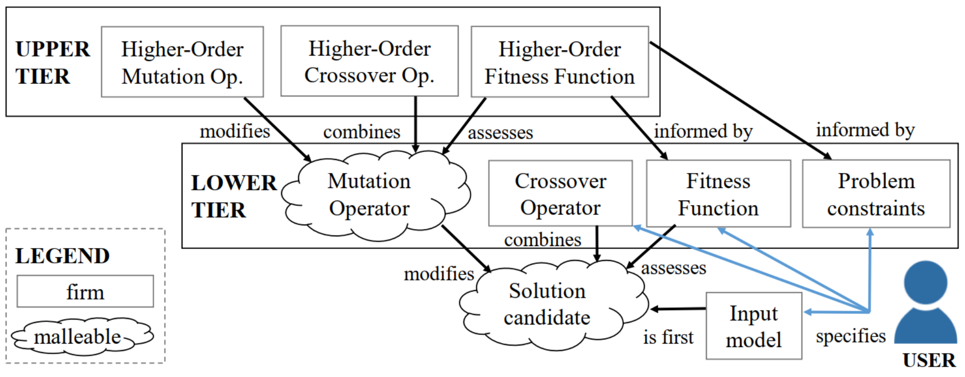


Figure 2.3: An overview of the FitnessStudio framework.

to generate mutation operators for the NRP.

The lower tier of the framework makes use of a genetic algorithm to optimize a set of solution candidates towards a fitness function. It requires specification of a crossover operator, fitness function and possible problem constraints, just like other search-based approaches. However, instead of having to specify the mutation operators, these are generated by the upper tier of the framework.

The upper tier is a search-based approach to generate efficient mutation operators. An initial set of mutation operators is optimized so that running the lower tier of the framework using that set of mutation operators results in the best model, assessed by the lower tier fitness function. The upper tier is generic, it does not need to be changed for different problems.

The original version of FitnessStudio was evaluated on a single-objective case and because the lower tier of the framework made use of a single-objective genetic algorithm, it could not be used for multi-objective problems. Therefore, we make the necessary modifications to use the multi-objective GA NSGA-II instead. Besides, the original version assumed that mutation operators are entirely generated from scratch, which does not allow to benefit from initial manually crafted solutions. Furthermore, all higher-order mutation rules are assumed to be equally useful, which we did not find to be the case for the NRP in our exploratory experiments. We address these two issues in Sec. 3.2.

Chapter 3

Single-Objective Implementation of the NRP

3.1 The NRP as a single-objective problem

The next release problem has two objectives: maximizing customer satisfaction and minimizing cost. However, we want to see whether generated mutation operators using FitnessStudio can improve single-objective results for the next release problem for two reasons. First, we can use FitnessStudio without having to re-implement the lower tier of the framework. Second, in some cases it can be useful to get a single result which is a good trade-off between different objectives [3]. Therefore, we specify the NRP as a single-objective optimization problem, combining the two objectives in the following way:

$Fit(s) = \frac{sat(s)}{max\_sat} - \frac{cost(s)}{max\_cost}$

Where max_sat and max_cost are the customer satisfaction and artifact cost if all software artifacts are selected for the problem instance and $sat(s)$ and $cost(s)$ are the satisfaction and cost of the selected artifacts of candidate solution s. For calculating $satisfaction$ and $cost$ we use the implementation from the MDEOptimiser project, adapted to address an implementation bug.

3.2 Alterations made to FitnessStudio

Combining the available and our generated mutation operator. There already exists a clever rule-based mutation operator for the NRP provided by MDEOptimiser [6]. So, instead of ignoring that set, we combine it with our generated mutation operators using FitnessStudio with the aim of improving performance. As can be seen in Fig 3.1, each iteration the lower tier randomly selects either the fixed ruleset or our generated ruleset with equal chance. This technique leads to a configuration option called fixedXORgen which we will compare against fixed: a configuration option that only uses the available mutation operator.

```
if (applyFixedRules)
   // LowerTierRunner
   if (Math.random() > 0.5)
      mutateWithFixedRules(graph);
   else
      mutateWithGenRules(graph);
else
   // UpperTierRunner
   mutateWithGenRules(graph);
```

Figure 3.1: fixedXORgen.

Higher-order mutation rule weight. FitnessStudio applies all higher-order mutation rules with a fixed chance. However, for the NRP, not every higher-order mutation rule is equally useful. First, no node should be added or deleted from the domain model. Subsequently, the mutation rule $createCrOrDelNodeWithContainmentEdge$ can best be given no weight. Second, a rule that does not create or delete an edge between Solution and Software Artifact will not change the fitness score. Testing showed that increasing the weight of the mutation rule $createCrOrDelEdge$ improved performance. Therefore, we introduce a map in which the weight of each higher-order mutation rule can be tuned individually.

3.3 Implementation

We implemented the next release problem as a single-objective problem using FitnessStudio including the alterations discussed in Sec. 3.2, providing the implementation at [31].

FitnessStudio uses a simple single-objective GA available at `https://github.com/lagodiuk/genetic-algorithm`. For the NRP implementation, we use single-point crossover with a probability of Pc = 0.9. This crossover operator is frequently used as component of genetic algorithms to solve the NRP [35, 15]. Each mutation rule is executed with a probability of Pm = 0.6.

3.4 Initialization

We experienced that there is not a single best way of performing initialization for our implementation of the NRP. Therefore, we compared some

	Empty				**Complete**				**Random**			
Input model	NRP best	NRP median	NRP stdev	Time median	NRP best	NRP median	NRP stdev	Time median	NRP best	NRP median	NRP stdev	Time median
A	0.407	0.353	0.022	00:12.1	**0.457**	**0.446**	**0.013**	**00:10.0**	0.454	0.439	0.017	00:11.4
B	0.408	0.347	0.020	00:39.9	0.526	0.504	**0.013**	**00:35.3**	**0.589**	**0.554**	0.023	00:38.4
C	0.311	0.279	**0.012**	01:04.0	0.379	0.357	**0.012**	**00:47.8**	**0.508**	**0.461**	0.025	00:59.2
D	0.418	0.399	**0.007**	01:26.3	0.314	0.276	0.010	**01:04.3**	**0.434**	**0.403**	0.019	01:20.9
E	0.198	0.145	0.015	02:11.9	0.218	0.207	**0.007**	**01:48.6**	**0.272**	**0.226**	0.023	02:15.0

Table 3.1: Results using the MDEOptimiser mutation operator, times denoted in mm:ss:x.

basic initialization techniques using the genetic algorithm and the available MDEOptimiser mutation operator.

Empty solutions contain no software artifacts, *complete* solutions contain all artifacts and for *random* solutions, each artifact has a 50% chance to be included in the solution. We also explored populations using combinations of the methods mentioned above as well as Extreme Solutions with Path Relinking (EPR, [11]). However, these methods did not seem worth analysing further.

Given the results from Table 3.1, we decided to include both complete and random initialization in our further evaluation. Random initialization seems to perform best for larger models and/or fewer iterations while complete initialization seems to work best for small models and/or more iterations and seems to have a smaller standard deviation than *random* initialization as well as the smallest run time.

3.5 Preliminary evaluation

Just as Strüber, 2017 [29], we investigated the following two research questions to evaluate the efficiency of combining available mutation operators with our generated mutation operators for the NRP: (**RQ1**) What is the quality of the solutions produced by combining the available and generated mutation operators for the NRP? (**RQ2**) How does combining the available and generated mutation operators affect performance for the NRP?

Scenario. In our evaluation we use five input models of varying size. Input models A and B were obtained from the MDEOptimiser project. We generated input models C-E using the MDEOptimiser NRPModelGenerator. All models were preprocessed to address an implementation bug discovered during work on this thesis. A brief description of these models is provided in Table 3.2.

Input models	**A**	**B**	**C**	**D**	**E**
Customers	5	25	50	75	100
Requirements	25	50	75	100	120
Artifacts	63	203	319	425	602

Table 3.2: Input models

To discuss the quality and performance of our technique that combines the available and our generated mutation operators as described in Sec. 3.2, we compare it to only using the available mutation operator. As metrics we use the median and best result of each run as well as the median run time.

Set-up. We generate the mutation operator for each initialization method using the smallest model A and reuse it for all models to limit overhead. All the models used and output generated for our experiments are available at [31].

Preparation: Generation of mutation operators. To create our own mutation operators, we applied FitnessStudio to input model A two times.

The first time, we generated mutation operators using *complete* initialization, configuring the upper tier to a population size of sixty and fifteen iterations with each a timeout of ninety seconds, and the lower tier to a population size of two and twenty iterations. We repeated the generation ten times; the median run took 13:28 minutes. The best-performing mutation operator produced by FitnessStudio contained five rules. One did not create/delete any edges and as a result, applying that rule would not change the candidate solution and its fitness score. One rule deleted two selectedArtifacts edges and the other three rules deleted both two solutions edges.

Next, we generated mutation operators using *random* initialization, configuring the upper tier to a population size of sixty and fifteen iterations with each a timeout of ninety seconds, and the lower tier to a population size of ten and ten iterations. We repeated the generation ten times; the median run took 10:08 minutes. The best-performing mutation operator produced by FitnessStudio contained four rules. One did not create/delete any edges and as a result, applying that rule would not change the candidate solution and its fitness score. One rule deleted a selectedArtifacts and a solutions edge and the remaining two rules deleted a single selectedArtifacts edge and solutions edge respectively.

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 thirty times, using a population size of forty and 120 iterations. All experiments where performed on a Windows 10 system (Intel Core i7-8850H, 2.6 GHz; 32 GB of RAM, Java 1.8 with 2 GB maximum memory size).

Results. Table 3.3, Table 3.4 and Fig. 3.2 show the results of our experiments.

RQ1: Result quality. For *complete* initialization, combining our generated mutation operator with the MDEOptimiser mutation operator improved the result quality for all five models. For *random* initialization, combining the two mutation operators improved the result quality for the larger three models significantly. For model A, result quality was slightly better using the combination of the two. But for model B result quality was slightly better using only the MDEOptimiser mutation operator. It seems that using our technique result quality improves most for larger models and/or fewer iterations.

	MDEOptimiser (fixed)			**Combi (fixedXORgen)**		
Input model	NRP best	NRP median	Time median	NRP best	NRP median	Time median
A	**0.457**	0.446	00:10.0	**0.457**	**0.457**	**00:08.2**
B	0.526	0.504	**00:35.3**	**0.582**	**0.556**	00:40.7
C	0.379	0.357	00:47.8	**0.459**	**0.435**	**00:43.3**
D	0.314	0.276	**01:04.3**	**0.427**	**0.405**	01:35.8
E	0.218	0.207	**01:48.6**	**0.357**	**0.336**	02:25.1

Table 3.3: Results using *complete* initialization, times denoted as mm:ss:x.

	MDEOptimiser (fixed)			**Combi (fixedXORgen)**		
Input model	NRP best	NRP median	Time median	NRP best	NRP median	Time median
A	0.454	0.439	00:11.4	**0.461**	**0.440**	**00:08.6**
B	0.589	**0.554**	00:38.4	**0.602**	0.540	**00:30.0**
C	0.508	0.461	00:59.2	**0.539**	**0.491**	**00:46.4**
D	0.434	0.403	01:20.9	**0.473**	**0.443**	**01:06.8**
E	0.272	0.226	02:15.0	**0.330**	**0.290**	**01:39.1**

Table 3.4: Results using *random* initialization, times denoted as mm:ss:x.

RQ2: Performance. Using *random* initialization, the median run time significantly decreased in all cases when using our technique of combining the two mutation operators. Using *complete* initialization, our technique decreased run time for models A and C while the run time significantly increased for the other three models. The generated mutation operator for *random* initialization seems to be more efficient than the one for *complete* initialization.

We suspect that the performance of generated mutation operators can be guided by limiting the difference between the run time of the first iteration and the timeout for the upper tier of FitnessStudio. Because a long run time for execution of an iteration of the upper tier of the framework seems to indicate a longer run time for the resulting mutation operator.

Limitations and Threats to Validity. We see one main limitation of our technique. Generating the mutation operator using the upper tier of the

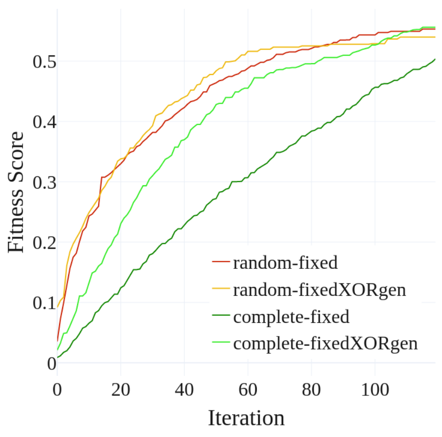


Figure 3.2: Model B, sixteenth best run over time.

framework takes time and is required before execution of the lower tier that actually finds solutions. The impact of this initial overhead depends on how often the lower tier is ran using the same generated mutation operator.

The major threat to external validity of our results is the limited scope of our experiments. We examined one benchmark case using five models of which the smallest two are provided by MDEOptimiser and the larger three are generated by the same model generator, also provided by MDEOptimiser. In real word cases it might not be that a mutation operator generated using a small model would be able to be used efficiently and effectively for larger models. Besides, we combined the fitness functions in a specific way to create a single-objective implementation. Using a different fitness function, results could be different. However, the technique is also evaluated for another, single-objective, benchmark case showing promising results [29].

Chapter 4

Multi-Objective Implementation of the NRP

4.1 Preliminaries

A multi-objective optimization problem most likely does not have a single best solution. In that case there will always be a trade-off between the different objectives. Therefore, as multi-objective solution, a set of solutions that are not dominated by any other solution is often used. A solution is dominates another if:

1) for at least one objective, the value of the fitness function is the better for the dominating than the dominated solution and
2) for all other objectives, the fitness function value is not worse.

A solution is Pareto optimal if it is not dominated by any solution in the solution space and can only be improved for an objective by worsening at least one other [23]. The set of all Pareto optimal solutions is called the Pareto optimal set. However, in practice, the so-called best-known Pareto front is used that consists of all non-dominated solutions that were found when searching. This is because evaluating and comparing the entire solution space is normally impossible. Otherwise, one could use random search and would not need to use genetic algorithms.

To compare the quality of two Pareto fronts for the same model, we will use the hypervolume (HV) and spread measure. Hypervolume is a widely used volume-based quality indicator and evaluates the optimizer outcome by simultaneously taking into account the proximity of the points to a given reference Pareto front, diversity, and spread [25]. And, even though the hypervolume measure takes spread into account, we also include spread as separate indicator. The hypervolume measure can be similar for two Pareto fronts where the first has a narrow spread with close proximity to the real/reference Pareto front and the second has a large spread but is further away from the reference Pareto front.

4.2 Implementation

We implemented the next release problem as a multi-objective problem using FitnessStudio including the alterations discussed in Section 3.2. We re-implemented the lower tier to use the multi-objective genetic algorithm NSGA-II [12] as implemented by the JMetal framework [14]. Our implementation is available at [31]. We replaced the original fitness evaluation for the upper tier with a fitness evaluation based on the hypervolume relative to the reference Pareto front. We obtained this reference front by executing the lower tier with a great number of evaluations and a large population, see Sec. 4.4. For the NRP implementation, we use single-point crossover as crossover operator with a probability of Pc = 0.9 and execute each mutation rule with a probability of Pm = 0.6.

4.3 Initialization

Like the single-objective implementation, the initialization method has a significant impact on the results of the multi-objective implementation. Because of that, we use different initialization methods for evaluation.

Empty solutions contain no software artifacts, *complete* solutions contain all artifacts, an *extremes* solution consists of alternating *empty* and *complete* solutions, for *random* solutions each artifact has a 50% chance to be included in the solution and $rand + x$ solutions consist of one *empty*, one *complete* solution and for the remainder of *random* solutions.

4.4 Preliminary evaluation

Just as for the single-objective implementation, we investigated the following two research questions to evaluate the efficiency of combining available mutation operators with our generated mutation operators for the NRP: (**RQ1**) What is the quality of the solutions produced by combining the available and generated mutation operators for the NRP? (**RQ2**) How does combining the available and generated mutation operators affect performance for the NRP?

Scenario. In our evaluation we use the same five input models as for the single objective implementation. A brief description of these models can be found in Table 3.2.

To discuss the quality and performance of our technique that combines the available and our generated mutation operators as described in section 3.2, we compare it to only using either the available mutation operator or our generated mutation operators using FitnessStudio. As metrics we use the mean hypervolume relative to the reference Pareto front, lower is better, together with its standard deviation, the mean spread, lower is better, and its standard deviation and the mean run time and its standard deviation.

Set-up. All the models used and output generated for our experiments are available at [31].

Preparation: Generation of mutation operators. For each initialization method, we applied FitnessStudio to input model A, configuring the upper tier to a population size of sixty and fifteen iterations with each a timeout of 180 seconds, and the lower tier to a population size of eight and a maximum of two hundred evaluations. We repeated the generation five times; the median run time for each initialization method can be found in Table 4.1.

Initialization method	**Time** *Median*
Complete	23:12.7
Empty	06:00.3
Extremes	08:12.7
Random	07:30.6
Rand+x	10:02.9

Table 4.1: Median run time generation mutation operator in mm:ss:x.

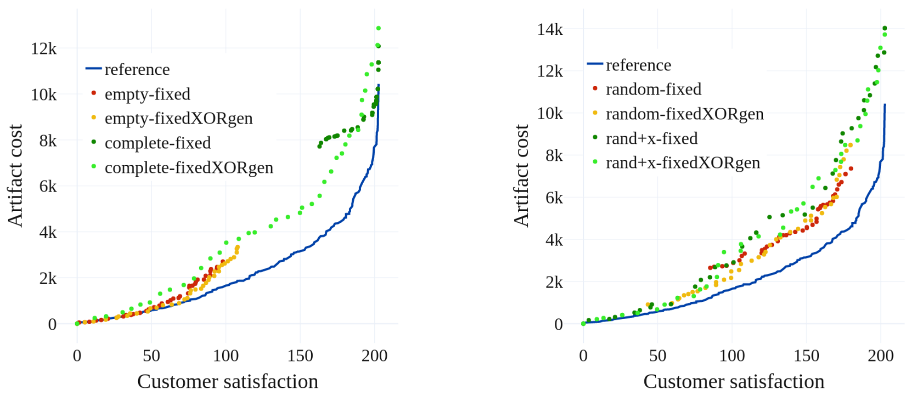
While testing, we experienced that using the ruleset produced by *extremes* initialization outperformed $rand + x$ initialization when running the lower tier using $rand{+}x$ initialization. Therefore, we ignore the $rand + x$ ruleset and use the *extremes* ruleset instead for evaluation using $rand + x$ initialization. We suspect that the *extremes* mutation operator is of higher quality while the random element when generating that ruleset is smaller than for the $rand{+}x$ ruleset.

Reference Pareto front. We compare all results to a reference Pareto front for each model. These are created using lower tier execution with the available mutation operator and $rand{+}x$ as initialization method. We ran it once for each model using a population size of two hundred and a maximum of 150,000 evaluations.

Benchmark measurement. For each initialization method, we applied the genetic algorithm together with the top-scoring generated mutation operator for that initialization method to models A-E. Except $rand + x$ for which we used the *extremes* mutation operator. To study the variability of the produced results, we repeated the experiment thirty times, using a population size of forty and a maximum of 5000 evaluations. All experiments where performed on a Windows 10 system (Intel Core i7-8850H, 2.6 GHz; 32 GB of RAM, Java 1.8 with 2 GB maximum memory size).

Results. Tables 4.2, 4.3 and 4.4 and Fig. 4.1 show the results of our experiments.

RQ1: Result quality. For model B, see Fig. 4.1 and Table 4.2, combining the MDEOptimiser mutation operator with our generated mutation operator results for four out of five initialization methods in an improved hypervolume and in four out of five cases in an improved spread. The *extremes* initialization method results in a slightly worse hypervolume and the $rand{+}x$ initialization method in a marginally worse spread. However, in both cases, the other quality measure is slightly improved. So, the result quality for model B is improved in three out of five cases and results are similar for



(a) *empty* and *complete* (b) *random* and $rand + x$

Figure 4.1: Model B, Pareto fronts of fifteenth best run.

the other two. Using *random* initialization, see Table 4.3, the result quality when combining the generated mutation operator and the available one is worse for model A. For all larger models, combining the two mutation operators results in an improved result quality. From Fig. 4.1(b), it seems like our generated mutation operator mostly improves the quality for solution sets having a smaller artifact cost. Using $rand + x$ initialization, see Table, 4.4, the result quality when using the combination of the generated and available mutation operator is improved for models A, C and D, is very similar for model B and is worse for model E. It could be that it cannot be expected that mutation operators generated for an initial population containing both extremes perform similarly for different models.

	MDEOptimiser (fixed)			**Combi (fixedXORgen)**			**FitnessStudio (gen)**		
Initialization	**HV**	**Spread**	**Runtime**	**HV**	**Spread**	**Runtime**	**HV**	**Spread**	**Runtime**
Complete	0.6967	0.9358	00:21.951	0.1853	**0.4311**	00:20.488	0.2011	0.4386	00:14.095
	(0.0414)	(0.0314)	(00:00.546)	(0.019)	**(0.0536)**	(00:00.398)	(0.0195)	(0.0389)	(00:00.364)
Empty	0.4125	0.7319	00:32.047	0.3926	0.705	00:21.658	0.4109	0.714	00:11.999
	(0.073)	(0.058)	(00:00.718)	(0.0271)	(0.0329)	(00:00.428)	(0.035)	(0.0489)	(00:00.257)
Extremes	0.2278	0.468	00:27.213	0.231	0.4649	00:21.131	0.2366	0.4769	00:11.731
	(0.0161)	(0.0424)	(00:00.315)	(0.0215)	(0.0493)	(00:01.417)	(0.0127)	(0.0445)	(00:00.291)
Rand+x	0.222	0.4697	00:29.022	0.2102	0.4708	00:20.806	0.2066	0.4814	00:12.904
	(0.0193)	(0.04)	(00:00.339)	(0.0179)	(0.0428)	(00:00.401)	(0.0248)	(0.0433)	(00:00.311)
Random	0.267	0.7146	00:28.057	**0.178**	0.5283	00:23.433	0.2887	0.6752	**00:11.356**
	(0.0267)	(0.0737)	(00:00.399)	**(0.0255)**	(0.0488)	(00:02.271)	(0.0388)	(0.046)	**(00:00.314)**

Table 4.2: Results for model B (mean, (stdev)), times denoted as mm:ss.xxx.

Input	MDEOptimiser (fixed)			Combi (fixedXORgen)		
model	**HV**	**Spread**	**Runtime**	**HV**	**Spread**	**Runtime**
A	**0.0911**	**0.5795**	00:08.817	0.1293	0.6642	**00:07.284**
	(0.0198)	**(0.0824)**	(00:00.335)	(0.0425)	(0.059)	**(00:00.486)**
B	0.267	0.7146	00:28.057	**0.178**	**0.5283**	**00:23.433**
	(0.0267)	(0.0737)	(00:00.399)	**(0.0255)**	**(0.0488)**	**(00:02.271)**
C	0.3512	0.8471	00:45.160	**0.2622**	**0.7381**	**00:33.351**
	(0.024)	(0.0479)	(00:00.830)	**(0.0249)**	**(0.0584)**	**(00:01.130)**
D	0.3985	0.8714	01:01.851	**0.3351**	**0.844**	**00:46.192**
	(0.0209)	(0.0494)	(00:01.004)	**(0.0257)**	**(0.045)**	**(00:01.139)**
E	0.4778	0.9129	01:34.769	**0.4672**	**0.9028**	**01:08.612**
	(0.0207)	(0.0303)	(00:01.455)	**(0.0226)**	**(0.0324)**	**(00:01.396)**

Table 4.3: *random* results (mean, (stdev)), times denoted as mm:ss:xxx.

Input	MDEOptimiser (fixed)			Combi (fixedXORgen)		
model	**HV**	**Spread**	**Runtime**	**HV**	**Spread**	**Runtime**
A	0.103	0.5079	00:08.986	**0.0704**	**0.501**	**00:06.342**
	(0.0134)	(0.0559)	(00:00.363)	**(0.0116)**	**(0.0544)**	**(00:00.282)**
B	0.222	**0.4697**	00:29.022	**0.2102**	0.4708	**00:20.806**
	(0.0193)	**(0.04)**	(00:00.339)	**(0.0179)**	(0.0428)	**(00:00.401)**
C	0.2629	0.4205	00:45.694	**0.2304**	**0.4297**	**00:31.953**
	(0.0163)	(0.0341)	(00:00.529)	**(0.021)**	**(0.0387)**	**(00:00.505)**
D	0.2415	0.4439	01:03.647	**0.2283**	**0.4314**	**00:43.879**
	(0.0131)	(0.0278)	(00:01.143)	**(0.0126)**	**(0.037)**	**(00:00.487)**
E	**0.3073**	**0.3698**	01:34.197	0.3277	0.3819	**01:06.508**
	(0.0151)	**(0.047)**	(00:01.501)	(0.0178)	(0.0574)	**(00:01.770)**

Table 4.4: $rand + x$ results (mean, (stdev)), times denoted as mm:ss:xxx.

RQ2: Performance. The run time decreased significantly for all tested models and initialization methods when combining our generated mutation operators with the one from MDEOptimiser. As a result, we could run NSGA-II with more evaluations for the same run time when combining our generated mutation operator with the available one. These results show that our technique can be used to improve performance relative to current mutation operators without sacrificing result quality.

Limitations and Threats to Validity. Like the single-objective implementation, the main limitation of our technique is that generating the mutation operator using the upper tier of the framework takes time and is required before execution of the lower tier that actually finds solutions. The impact of this initial overhead depends on how often the lower tier is ran using the same generated mutation operator.

The major threat to external validity of our results is the limited scope of our experiments. We examined one benchmark case using five models of which A and B were provided by and C-E generated by the model generator from MDEOptimiser. In real word cases it might not be that a mutation operator generated using a small model would be able to be used efficiently and effectively for larger models.

Chapter 5

Related Work

The next release problem is studied in a multitude of papers. The different approaches can be distinguished in terms of whether they represent the problem using vector encoding or using a model, like our technique.

Using vector encoding, multiple solutions exist. Most research uses meta-heuristics to solve the NRP. These meta-heuristics include GA's like NSGA-II [15, 35] and MOCell [15], and (hybrid) ant colony optimization [20, 13]. Besides meta-heuristics, more exact techniques like Integer Linear Programming (ILP) are to a lesser extend also used to solve the NRP. [32] found that a modern ILP solver can solve large single-objective and small multi-objective instances very quickly. However, it takes several hours for larger multi-objective instances.

For model representation, we are aware of two solutions for the NRP. The first uses models in combination with NSGA-II but without an automatic generated mutation operator and with a different model to represent the NRP [10]. The second [8] is the closest related to our research and does generate mutation operators. However, instead of meta-learning to generate mutation operators, the technique requires the user to specify the sub-metamodel for which mutation operators should be generated. An evaluation that compares the result quality and efficiency using these two techniques would be interesting.

Mutation generation. Our work expands the original single-objective version of the FitnessStudio framework [29] that generates mutation operators for MDE using meta-learning to an implementation for the multi-objective next release problem. Apart from this technique, we distinguish two other techniques to create mutation operators for search-based model-driven engineering. Firstly, the technique used in the paper discussed in the previous paragraph by Burdusel et al. [8]. They generalise earlier work [7] where for a single case study, initial ideas for generating atomic mutations were explored. It also builds on the work in [2] and [26], which uses atomic mutations as well for automating the mutation operator. Secondly, Wodel [16] can be used to create mutation operators. Wodel is a domain-specific language and tool for model-based mutation that is independent of the domain meta-model. It can aid in the creation of mutation operators but still requires the user to specify the mutation rules. Beyond generating mutation operators, another research direction is on analysing generated operators [24], to check whether these operators contribute to improving the consistency of solutions or at least do not introduce new violations of well-formedness constraints.

Outside of MDE, mutation generation is also a subject of research. Self-adaptive mutation [27] adapts the mutation probability during execution of the genetic algorithm. Besides, the generation of mutation operators is studied in [34] and [18]. The first uses Register Machines to explore a constrained design space for mutation operators and the second genetic programming to generate mutation operators. Both use meta-learning to create the mutation operator.

Chapter 6

Conclusions and Future Work

The goal of this research is to explore whether generated rule-based mutation operators can be used to for solving multi-objective problems using model representation. We combined our generated mutation operators with an available mutation operator for the next release problem and compared result quality and performance to using only the available mutation operator for both a single- and multi-objective implementation of the NRP. Our results are promising for both implementations showing improved result quality as well as performance in most cases. Our research also shows the impact different initialization methods have on the results.

In our research we encountered three possible directions for future work. First, the timeout parameter when generating mutation operators might have an influence on the efficiency of the generated mutation operator (Sec. 3.5). Limiting timeout further might increase efficiency of our technique. As second direction for future work one could investigate how initialization methods can best be chosen when using our technique. Because our research shows that the used initialization method can have a significant influence on results both when generating mutation operators as well as applying them. Finally, it would be interesting to directly compare result quality and performance of our technique to using atomic mutation operators [8].

Besides our directions, the three directions for future work on FitnessStudio mentioned by Strüber, 2017 [29], still apply. These are optimizing the the higher-order mutation; applying the principles of the technique for the generation of cross-over operators; and application to a broader variety of use-cases.

Bibliography

[1] Eclipse Modeling Framework (EMF). `https://www.eclipse.org/modeling/emf/`. [Online; accessed 2021-06].

[2] Faisal Haji M Alhwikem, Richard Freeman Paige, Louis Matthew Rose, and Robert David Alexander. A systematic approach for designing mutation operators for mde languages. 2016.

[3] Vahid Alizadeh, Houcem Fehri, and Marouane Kessentini. Less is more: From multi-objective to mono-objective refactoring via developer's knowledge extraction. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 181–192. IEEE, 2019.

[4] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: advanced concepts and tools for in-place emf model transformations. In *International Conference on Model Driven Engineering Languages and Systems*, pages 121–135. Springer, 2010.

[5] Anthony J. Bagnall, Victor J. Rayward-Smith, and Ian M Whittley. The next release problem. *Information and software technology*, 43(14):883–890, 2001.

[6] Alex Burdusel and Steffen Zschaler. Next Release Problem - MDEOptimiser. `https://mde-optimiser.github.io/case-studies/nrp/`, 2017. [Online; accessed 2021-06].

[7] Alexandru Burdusel and Steffen Zschaler. Towards automatic generation of evolution rules for model-driven optimisation. In *Pre-Proceedings of the 8th International Workshop on Graph Computation Models (GCM'17)*, 2018.

[8] Alexandru Burdusel, Steffen Zschaler, and Stefan John. Automatic generation of atomic consistency preserving search operators for search-based model engineering. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 106–116. IEEE, 2019.

[9] Alexandru Burdusel, Steffen Zschaler, and Daniel Strüber. Mdeoptimiser: A search based model engineering tool. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 12–16, 2018.

[10] Frank R Burton, Richard F Paige, Louis M Rose, Dimitrios S Kolovos, Simon Poulding, and Simon Smith. Solving acquisition problems using model-driven engineering. In *European Conference on Modelling Foundations and Applications*, pages 428–443. Springer, 2012.

[11] Thiago Gomes Nepomuceno Da Silva, Leonardo Sampaio Rocha, and José Everardo Bessa Maia. An effective method for mogas initialization to solve the multi-objective next release problem. In *Mexican International Conference on Artificial Intelligence*, pages 25–37. Springer, 2014.

[12] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In *International conference on parallel problem solving from nature*, pages 849–858. Springer, 2000.

[13] José Del Sagrado, Isabel María Del Águila, and Francisco Javier Orellana. Ant colony optimization for the next release problem: A comparative study. In *2nd International Symposium on Search Based Software Engineering*, pages 67–76. IEEE, 2010.

[14] Juan J Durillo and Antonio J Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760–771, 2011.

[15] Juan J Durillo, Yuanyuan Zhang, Enrique Alba, Mark Harman, and Antonio J Nebro. A study of the bi-objective next release problem. *Empirical Software Engineering*, 16(1):29–60, 2011.

[16] Pablo Gómez-Abajo, Esther Guerra, Juan de Lara, and Mercedes G Merayo. A tool for domain-independent model mutation. *Science of Computer Programming*, 163:85–92, 2018.

[17] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.

[18] Libin Hong, John H Drake, John R Woodward, and Ender Özcan. A hyper-heuristic approach to automated generation of mutation operators for evolutionary programming. *Applied Soft Computing*, 62:162–175, 2018.

[19] Hisao Ishibuchi, Yusuke Nojima, and Tsutomu Doi. Comparison between single-objective and multi-objective genetic algorithms: Performance comparison and performance measures. In *2006 IEEE International Conference on Evolutionary Computation*, pages 1143–1150. IEEE, 2006.

[20] He Jiang, Jingyuan Zhang, Jifeng Xuan, Zhilei Ren, and Yan Hu. A hybrid aco algorithm for the next release problem. In *The 2nd International Conference on Software Engineering and Data Mining*, pages 166–171. IEEE, 2010.

[21] Stefan John, Alexandru Burdusel, Robert Bill, Daniel Struber, Gabriele Taentzer, Steffen Zschaler, and Manuel Wimmer. Searching for optimal models: Comparing two encoding approaches. In *12th International Conference on Model Transformations ICMT 2019*, pages 1–22, 2019.

[22] Stuart Kent. Model driven engineering. In *International conference on integrated formal methods*, pages 286–298. Springer, 2002.

[23] Abdullah Konak, David W Coit, and Alice E Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability engineering & system safety*, 91(9):992–1007, 2006.

[24] Jens Kosiol, Daniel Strüber, Gabriele Taentzer, and Steffen Zschaler. Graph consistency as a graduated property. In *International Conference on Graph Transformation*, pages 239–256. Springer, 2020.

[25] Miqing Li and Xin Yao. Quality evaluation of solution sets in multiobjective optimisation: A survey. *ACM Computing Surveys (CSUR)*, 52(2):1–38, 2019.

[26] JGM Mengerink, Alexander Serebrenik, Ramon RH Schiffelers, and MGJ Van Den Brand. A complete operator library for dsl evolution specification. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 144–154. IEEE, 2016.

[27] Daniel Smullen, Jonathan Gillett, Joseph Heron, and Shahryar Rahnamayan. Genetic algorithm with self-adaptive mutation controlled by chromosome similarity. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 504–511. IEEE, 2014.

[28] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[29] Daniel Strüber. Generating efficient mutation operators for search-based model-driven engineering. In *International Conference on Theory and Practice of Model Transformations*, pages 121–137. Springer, 2017.

[30] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. Henshin: A usability-focused framework for emf model transformation development. In *ICGT'17: International Conference on Graph Transformation*, pages 196–208. Springer, Cham, 2017.

[31] Niels van Harten. Fitnessstudio applied on the next release problem. `https://doi.org/10.6084/m9.figshare.14774448`, 2021.

[32] Nadarajen Veerapen, Gabriela Ochoa, Mark Harman, and Edmund K Burke. An integer linear programming approach to the single and bi-objective next release problem. *Information and Software Technology*, 65:1–13, 2015.

[33] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[34] John R Woodward and Jerry Swan. The automatic generation of mutation operators for genetic algorithms. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 67–74, 2012.

[35] Yuanyuan Zhang, Mark Harman, and S Afshin Mansouri. The multi-objective next release problem. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1129–1137, 2007.