Model-Driven Optimization: Generating Smart Mutation Operators for Multi-Objective Problems

Niels van Harten
Radboud University
Nijmegen, Netherlands
niels@vharten.com

Carlos Diego N. Damasceno
Radboud University
Nijmegen, Netherlands
d.damasceno@cs.ru.nl

Daniel Strüber
Chalmers | University of Gothenburg, SE
Radboud University Nijmegen, NL
danstru@chalmers.se

***Abstract*—In search-based software engineering (SBSE), the choice of search operators can significantly impact the quality of the obtained solutions and the efficiency of the search. Recent work in the context of combining SBSE with model-driven engineering has investigated the idea of automatically generating smart search operators for the case at hand. While showing improvements, this previous work focused on single-objective optimization, a restriction that prohibits a broader use for many SBSE scenarios. Furthermore, since it did not allow users to customize the generation, it could miss out on useful domain knowledge that may further improve the quality of the generated operators. To address these issues, we propose a customizable framework for generating mutation operators for multi-objective problems. It generates mutation operators in the form of model transformations that can modify solutions represented as instances of the given problem meta-model. To this end, we extend an existing framework to support multi-objective problems as well as customization based on domain knowledge, including the capability to specify manual "baseline" operators that are refined during the operator generation. Our evaluation based on the Next Release Problem shows that the automated generation of mutation operators and user-provided domain knowledge can improve the performance of the search without sacrificing the overall result quality.**

***Index Terms*—Model-Driven Engineering, Search-Based Software Engineering, Multi-Objective Optimization**

I. Introduction

Search-based software engineering (SBSE, [1]) seeks to solve software engineering problems using meta-heuristic techniques. One of the main classes of used techniques are genetic algorithms (GAs, [2]), which mimic the biological process of evolution. GAs improve an initial population of candidate solutions using search operators favoring reproduction of good over bad solutions. A SBSE solution to a given problem has three ingredients: (i) a problem encoding, (ii) a set of fitness functions to optimize, (3) a set of search operators, including mutation, crossover, and selection operators.

A present research line is on *model-driven optimization* (MDO, [3]), i.e., using model-driven techniques to bridge the abstraction gap between declarative specifications of optimization problems and low-level SBSE implementations. A key benefit is that developers can reuse available abstractions (models in the sense of MDE), instead of encoding candidate solutions as bit or integer vectors. This also allows developers to specify and experiment with genetic operators, such as mutation and crossover, more directly, as model transformations.

Developers of SBSE solutions face a considerable design space of search operators, ranging from naive, generic mutations (e.g., flipping a bit of a vector encoding [4]) to smart, problem-tailored operators (e.g., applying constraint solving within a mutation step to improve the solution [5], ensuring soundness by generating constraint-preserving mutation operators [6]). Without automated support, the design of search operators entirely relies on the intuition of the developer, which, could lead to plausible, yet inefficient solutions.

For example, consider the *Class Responsibility Assignment* problem (CRA, [7]), a critical benchmark in the current research line on MDO. CRA addresses the task of creating an optimal class design for a given set of methods and attributes. Our previous work [8] contributed an automated technique for generating *smart* mutation operators that outperformed nine previous solutions to CRA [9], which used manually crafted operators. The automatically generated operators performed a certain change that is likely to produce improved solutions—moving methods to classes that already contain a dependent method or attribute, thus improving coherence—, an idea that was not explored in any of the manually crafted solutions.

That previous automated technique, named *FitnessStudio* [8], produces efficient problem-tailored mutation operators based on a framework of two nested genetic algorithms: An upper-tier algorithm "tunes" the mutation operator of a lower-tier algorithm, where the latter is a conventional optimization run on an example problem instance. However, FitnessStudio has only been validated on CRA, a comparatively simple optimization scenario. We observe two main limitations that prohibit a wider application of this approach: (i.) *Lack of support for multi-objective optimization*. Whereas FitnessStudio is geared towards single-objective problems, many software engineering problems are indeed multi-objective problems that cannot be reasonably addressed with a single-objective approach, including *release planning* [10], *software product line testing* [11], *model transformation recovery* [12], *model merging* [13], and *design space exploration* [14]. Moreover, even for single-objective problems, multi-objective optimization can be used to improve the performance of the search (via helper objectives [15]), which is not possible in approaches that only support a single fitness function. (ii.) *Lack of support for user-provided domain knowledge*. Aiming to overcome the limitations of manual operator definition, FitnessStudio relied on an entirely automated generation process. Removing human intuition from the process lead to improved results in the CRA case. However, it remains open whether more complicated, multi-objective problems would benefit from user input that allows to draw from available domain knowledge.

In this paper, we present a technique for automatically generating efficient mutation operators for multi-objective search problems, in the context of model-driven optimization. We build on the FitnessStudio technique of nesting genetic algorithms in a two-tier framework, which we extend in two main directions: (i.) To support multi-objective optimization problems, we provide a new component for the evaluation of problem-specific mutation operators in the upper tier. This makes our technique applicable to the vast array of available multi-objective software engineering problems. (ii.) To support user-provided domain knowledge, we provide two modifications: First, the user can provide an initial mutation operator as input, which is then further tuned by the framework. Second, the user can fine-tune the upper tier using a configuration.

We evaluate our technique on the next release problem (NRP, [10]). NRP is a practically relevant problem of answering the following question: *What is the optimal subset of tasks to include in the next release of a company's product, to minimize development cost and maximize customer satisfaction?* As an NP-hard multi-objective optimization problem that, for large scale instances cannot be solved using exact optimization techniques [16], NRP has been used for evaluations of previous work [17] and is relevant for us as well.

The contributions of this paper are as follows:

- a technique for generating efficient mutation operators, based on a two-tier framework, supporting multi-objective problems and user-provided domain knowledge.
- an (from-scratch) implementation of this technique, based on the JMetal multi-objective optimization framework.
- an evaluation of this technique in the NRP case, in which we find improved performance when compared to an exclusively manually defined technique.
- a replication package containing all implementation and evaluation artifacts developed in the course of this work.

II. Background

We now introduce the necessary background. Since our work relies on model-driven engineering (MDE, [18]), specifically on model transformations, we introduce relevant background on MDE in an optimization context, before presenting the FitnessStudio technique [8], on which our new contribution builds. As an illustrative example, we consider the next release problem, which also forms the basis for our evaluation.

The next release problem. Any company developing and maintaining software products sold to a range of diverse customers faces the next release problem (NRP, [10]). The NRP is about determining what should be included in the next release of a product. 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, a business-critical decision-making task.

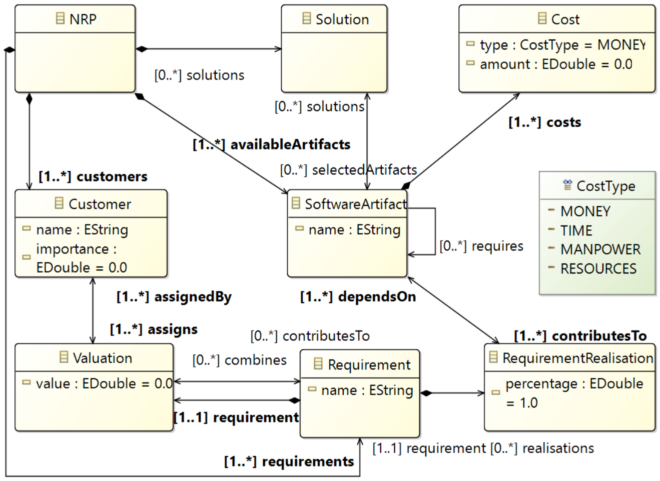


Fig. 1: Metamodel of the NRP case.

Previous work found that MDE solutions can function as a direct problem encoding for NRP [17], [19]. Figure 1 shows an available metamodel for the problem (from [19]). A model instance consists of a number of *solutions* where a solution contains a subset of the *availableArtifacts* called the *selectedArtifacts*. A software artifact has a *cost*, *contributesTo* a *RequirementRealisation*. An artifact may *require* 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 NRP has two objectives. Given above metamodel, the first is to minimize the total cost for the selected artifacts in a solution. The second 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. Cost and satisfaction are defined as follows:

$$Cost = \sum_{sa \in SA'} cost(sa)$$
$$Satisfaction = \sum_{c \in C} importance(c) \cdot satisfaction(c)$$

Here, *SA'* is the set of *selectedArtifacts*, *cost(sa)* is the cost for artifact *sa*, *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. A requirement can be realised directly in one or more ways depending on the available *realisations*. A *requirementRealisation* is fulfilled if all *softwareArtifacts* the realisation depends on and all their dependencies are included in the solution. If multiple realisations are fulfilled, the one with the highest percentage is chosen. A requirement can also be fulfilled by a combination of other requirements. The level of fulfillment is then determined by the weighted sum of the level of fulfillment of those dependencies.

For example, consider a Requirement A that 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 and the customer satisfaction are maximal. The goal is to find a good trade-off between maximizing customer satisfaction while minimizing the cost of all selected software artifacts.

Pareto-optimality. In multi-objective optimization problems such as NRP, there is usually a trade off between several different "best" solutions. One considers a *set of solutions* that only contains solutions that are not dominated by any other solution. A solution dominates another solution if performs better regarding at least one objective, while not performing worse regarding any other. 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 [20]. The set of all Pareto optimal solutions is called the Pareto optimal set. However, in practice, the so-called best-known Pareto set 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 [21]. In addition, we include spread in isolation as a 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.

Model transformation. Our solution uses the transformation language Henshin [22], [23]. Henshin is a rule-based language based on algebraic graph transformations and the Eclipse Modeling Framework (EMF, [24]), consisting of a tool set with editors and an interpreter engine. Using Henshin, we can specify *rules* for in-place model transformations. These rules specify basic "match and change" patterns. Our rationale for using Henshin is threefold: first, thanks to the declarative, graphical format of Henshin, the generated operators can be easily inspected; second, representing mutation operators in Henshin makes it easy to transform them systematically (via higher-order transformation rules); third, to support a fair performance comparison against the baseline, a solution from the MDEOptimser framework [19], using Henshin as well.

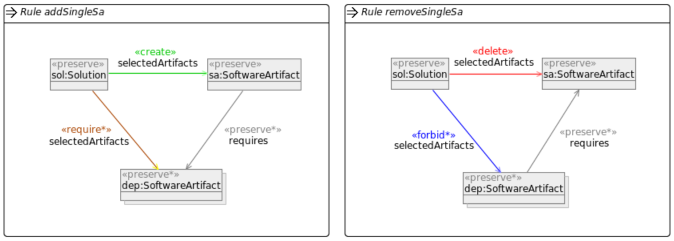


Fig. 2: Henshin rules specifying a mutation operator

Figure 2 shows rules from the mutation operator of the available MDEOptimiser solution of NRP. 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 these rules for mutation. While it is easy to come up with rules that should improve model quality at least in certain situations, it is hard to determine a generally efficient set of mutation rules.

FitnessStudio. FitnessStudio [8] is a two-tier framework for generating efficient mutation operators for SBSE problems. An overview of this framework is given in Fig. 3 (ignoring gray coloring for now). The lower tier uses a genetic algorithm to optimize a set of solution candidates towards a (single) fitness function. It requires specification of a crossover operator, fitness function and possible problem constraints, just like other search-based techniques. However, instead of expecting the user to specify the mutation operators, these are generated by the upper tier of the framework. The upper tier uses a genetic algorithm to generate efficient mutation operators for the lower tier. 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 remains constant over different problems.

FitnessStudio used a single-objective genetic algorithm on both tiers, which makes it inapplicable to multi-objective problems. Besides, the original version assumed that mutation operators are generated entirely automatically and from scratch, without additional user input, which does not allow to benefit from available domain knowledge.

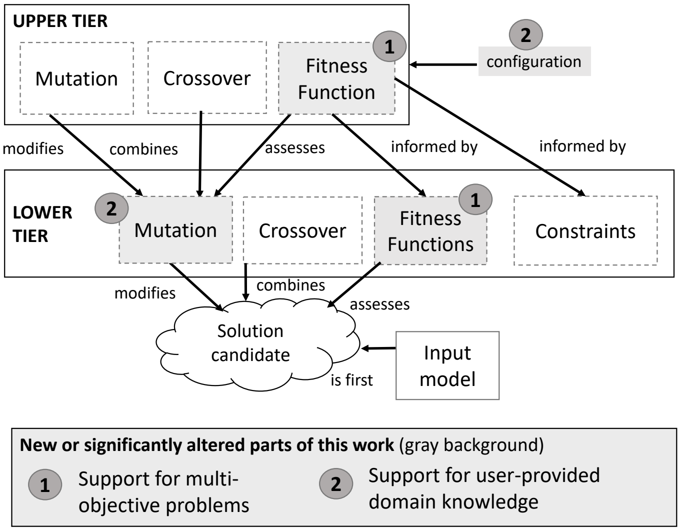


Fig. 3: The mutation operator generation technique following a two-tier framework [8] and several new components.

III. Proposed technique

The key contribution of this work is a technique for generating efficient mutation operators that, in contrast to earlier work it builds on, supports: (i) multi-objective optimization problems, (ii) user-provided domain knowledge. The technique follows the two-tier architecture of FitnessStudio, shown in Fig. 3 and explained in detail in Sect. II. To achieves its goals, it includes several new or significantly altered components, which we now discuss in detail.

A. Supporting multi-objective optimization problems

To support multi-objective problems, our technique needs to allow an arbitrary number of lower-tier fitness functions, instead of just one, as in single-objective problems. This raises the question of how the fitness of a generated mutation operator can be evaluated in the upper tier. The challenge is that the lower tier produces a Pareto front of solutions, instead of a single solution whose value can be used for easily pinning down the fitness of its mutation operator.

A tempting solution is to turn the upper-tier algorithm into a multi-objective one as well. To this end, one could represent each of the lower-tier objective functions by a corresponding upper-tier one. Values for the upper-tier objective functions to assess a concrete solution (lower-tier mutation operator) could be obtained from the training execution of the lower-tier algorithm. The drawback of this solution is that it will generally produce a pareto front of lower-tier mutation operators with no obvious guidance of how to select the single most appropriate solution among them.

Instead, in our solution, we strive to obtain a single best-performing mutation operator. To this end, we need a single value for comparison of mutation operators. We obtain this value by replacing the original upper-tier fitness function with one that computes the hypervolume (see Sect. II) of the solutions produced in the training execution of the lower tier. The hypervolume is computed relative to a reference Pareto front that we obtain by executing the lower tier with a great number of evaluations and a large population.

B. Supporting user-provided domain knowledge

Our support for user-provided domain knowledge is twofold: First, we allow the user to specify a part of the generated mutation operator manually. Second, we provide a means for configuring the upper-tier algorithm.

User-specified rule set. Like any program generation task, the fully automated generation of mutation operators from scratch is a hard problem. While FitnessStudio showed that full automation can lead to efficient solutions for a small-scale problem (CRA, comprising four concrete meta-classes), this might not necessary be the case for much more complicated problems, such as NRP. In our technique, we explore the idea that more efficient mutation operators could be obtained by having a user-specified fixed operator, which is further improved by combining it with automatically generated rules. This way, we strive for a "best of both worlds" solution of having a useful, but not necessarily optimal manually crafted mutation operator which is continuously improved by our technique.

```
if (Math.random() > 0.5)
    mutateWithFixedRules(graph);
else
    mutateWithGenRules(graph);
```

Listing 1: fixedXORgen: Combining generated with user-specified fixed rules

This rationale leads to the lower-tier mutation operator defined in Listing 1. In each iteration of the lower-tier algorithm, one rule set is randomly selected with equal chance, either the fixed or our generated ruleset. 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.

Configuration of upper-tier algorithm. In its upper-tier algorithm, FitnessStudio applies all higher-order mutation rules with a fixed chance. However, for different cases such as 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. To this end, we introduced a means to configure the upper-tier algorithm by assigning a weight to each higher-order mutation rule.

C. Implementation

While our framework follows the same overall structure as previous work [8], we implemented it from scratch, since the support for multiple objectives is a significant design decision that affects the entire implementation architecture. We implemented the framework based on the JMetal framework [25], which allows choosing between several different multi-objective genetic algorithms for the lower tier (e.g., NSGA-II [26], which has been widely used in other search-based software engineering studies and which we use as well). In the upper tier of our technique, we use a single-objective genetic algorithm. The framework, implemented in Java, is available as part of our replication package [27].

IV. Evaluation

We evaluated our technique on the next release problem (NRP), a challenging and practical problem described in detail in Sect. II. We instantiated our framework for NRP, described in Sect. III-B, and sought to answer two research questions:

- **RQ1**: How does the mutation operator generated by our framework impact performance, compared to a manually specified operator?
- **RQ2**: To which extent does the customization with user-provided domain knowledge impact the performance?

RQ1 addresses our framework's overall performance, measured in terms of the quality of the obtained solutions as well as execution time. RQ2 addresses the effect of one of our two contributions in isolation. To this end, we applied it to a single-objective version of our considered NRP case, so that obtained effect could be attributed to the customization alone.

A. Setup.

Applying our technique to a specific problem requires to provide the required input for the lower-tier components, as shown in Fig. 3 (whereas the upper-tier components are fixed). For the NRP case, we reused the crossover, fitness, and constraint implementations from the available MDEOptimiser implementation [19], whereas the mutation generator is generated by our approach. Details about parameter settings are provided in the Appendix at [27].

In our evaluation we use five input models (a.k.a. problem instances) of varying characteristics, as shown in Table I. Our first two input models, called A and B, were obtained from the MDEOptimiser project [19] and have been considered in earlier work on a performance comparison for the NRP case [17]. To obtain additional, larger input models, we used an available model generator provided by MDEOptimiser, NRPModelGenerator, to generate three additional models C-E.

The *baseline* for our performance comparisons is the available handcrafted mutation operator for the NRP case [19], which was found useful in an earlier performance comparison [17]. We also considered FitnessStudio as a potential baseline (encoding the two objectives of NRP into a single objective), but exploratory experiments showed that the resulting solutions were worse than those produced by the handcrafted mutation operator—hence, we excluded FitnessStudio from the comparison and the presentation of results in this paper.

TABLE I: Input models

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

In exploratory experiments, we experienced that there is no single best way of performing initialization for our scenario, that is, determining the initial assignment of artifacts to solutions. Therefore, we considered three basic initialization strategies: *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. In addition to these strategies, we also explored a combination of these strategies—*rand+x*, in which solutions consist of one *empty*, one *complete* solution and for the remainder of *random* solutions—, as well as one additional strategy: extreme solutions with path relinking (EPR, [28]). For both of our RQs, we report the best-performing of these strategies.

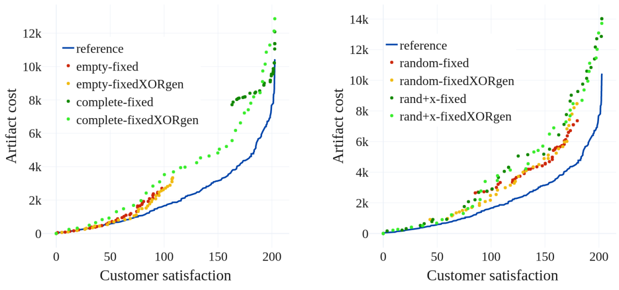
Our experiments where performed on a Windows 10 system (Intel Core i7-8850H, 2.6 GHz; 32 GB of RAM, Java 1.8 with 2 GB maximum memory size). All models used and output generated for our experiments are available at [27].

We further made the following detailed design decisions.

RQ1. We use standard evaluation metrics for our considered multi-objective problem: for result quality, the mean *hypervolume* (in short, *HV*; lower is better) and the mean *spread* (lower is better). Both metrics were introduced in Sect. II and are widely used in NRP and comparable problems. Hypervolume is calculated relative to an available reference pareto front, in our case created using lower-tier execution with the available baseline operator and *rand+x* as initialization method, using a population size of 200 and a maximum of 150K evaluations. For performance, we consider the execution time for a fixed number of runs. For all measurements, we report standard deviations obtained from the multiple experiment runs. Standard deviations allow to express the variability over multiple runs—a small standard deviation is desirable. We repeated the experiment thirty times, using a population size of forty and a maximum of 5K evaluations.

We performed the operator generation for each initialization strategy separately, providing details in the Appendix at [27]. The generated rules contained several recurring patterns. A prominent one is the creation or deletion of multiple *selectedArtifacts* edges between a solution and two software artifacts. This idea was not explored in the handcrafted operator, which focused on creating or deleting only one edge at one time—compare Fig. 2. The benefit of such a rule in the resulting operator is that it allows to make larger "steps" in the search space, potentially improving performance. Creating and deleting one *selectedArtifacts* edge at one time is a recurring pattern as well, with different rules specifying a varying amount of context (*preserve* elements of rules).

RQ2. The next release problem has two objectives: maximizing customer satisfaction and minimizing cost. For RQ2, we want to study the effect of our customization (one of the two main contributions of our work) in isolation and hence, needed a way to “factor out” the effect of the other main contribu-



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

Fig. 4: RQ1: Model B, Pareto fronts of 15-th best run (*empty*, *complete*, *random*, *rand+X* initialization); mutation operators from baseline (*fixed*) vs. our technique (*fixedXORgen*).

tion, that is, natively supporting multi-objective problems. We specify the NRP as a single-objective optimization problem, combining the two objectives as follows (with max_sat and max_cost representing customer satisfaction and artifact cost if all software artifacts are selected, and $sat(s)$ and $cost(s)$ as the satisfaction and cost in candidate solution s):

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

We performed the operator generation and the experiments for two selected initialization strategies, providing justification and details in [27]. Similarly to the rules generated for the multi-objective version in RQ1, those generated for RQ2 showed some recurring patterns: specifically, the deletion of two *selectedArtifacts* edges at the same time.

We applied the genetic algorithm together with the top-scoring generated mutation operator to models A-E. To study result variability, we repeated the experiment thirty times, using a population size of forty and 120 iterations.

B. Results for RQ1: Overall performance

Result quality. Table II gives an overview of our results. Using *random* initialization, for all models except for the smallest model, A, the mutation operator from our technique results in improved result quality. Using *rand+x* initialization (Table II, right), 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. Potentially, mutation operators generated for an initial population containing both extremes cannot be expected to perform similarly for different models. The picture for the additional initialization strategies (not shown) is similar: our technique in most cases produces somewhat better solutions than the baseline, but not consistently so for all models. In all cases, the observed standard deviations small, showing a good ability to produce consistent solutions over multiple runs.

For a more detailed analysis, we consider the results for model B, as illustrated in Fig. 4 (and in the Appendix available at [27]), 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 the other two. From Fig. 4b, it seems like our generated mutation operator mostly improves the quality for solution sets having a smaller artifact cost.

Execution time. The execution time decreased significantly for all tested models and initialization methods when combining our generated mutation operators with the one from MDEOptimiser. This observation can be explained by the higher complexity of the manually defined baseline operator, which relies on complicated control flow, leading to longer execution times. In the combined operator, using our generated rules, which are generally simple and therefore, easy to execute, improves the execution time. Similar to the quality measurements, we observe small standard deviations for both the baseline and our technique, showing a good ability to have similar execution times over multiple runs.

Summary of results (RQ1). The results indicate that our technique is able to support the generation of mutation operators for multi-objective problems [8]—leading to a benefit over previous work, which focused on single-objective problems. The generated operators can be used to improve the performance of the search (in terms of execution time) relative to hand-crafted mutation operators, without sacrificing result quality (in terms of hypervolume and spread).

C. Results for RQ2: Usefulness of customization

Result quality. Table III shows RQ2's results. For *complete* initialization, combining our generated mutation operator with the MDEOptimiser mutation operator improved the result quality for all five models. For *random* initialization, using the combined mutation operator improved the result quality for the larger three models significantly. For model A, result quality was slightly better using the combination of the two. For model B quality was slightly better using only the baseline mutation operator. It seems that using our technique quality improves especially for larger models or fewer iterations.

Runtime 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 execution 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 execution time of the first iteration and the timeout for the upper tier. This is 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.

TABLE II: RQ1: $random$ and $rand + x$ results (mean, (stdev)), times denoted as mm:ss:xxx.

Results	$random$						$rand + x$					
Input	Baseline (fixed)			Contribution (fixedXORgen)			Baseline (fixed)			Contribution (fixedXORgen)		
model	**HV**	**Spread**	**Runtime**	**HV**	**Spread**	**Runtime**	**HV**	**Spread**	**Runtime**	**HV**	**Spread**	**Runtime**
A	**0.0911**	**0.5795**	00:08.817	0.1293	0.6642	**00:07.284**	0.103	0.5079	00:08.986	**0.0704**	**0.501**	**00:06.342**
	(0.0198)	**(0.0824)**	(00:00.335)	(0.0425)	(0.059)	**(00:00.486)**	(0.0134)	(0.0559)	(00:00.363)	**(0.0116)**	**(0.0544)**	**(00:00.282)**
B	0.267	0.7146	00:28.057	**0.178**	**0.5283**	**00:23.433**	0.222	**0.4697**	00:29.022	**0.2102**	0.4708	**00:20.806**
	(0.0267)	(0.0737)	(00:00.399)	**(0.0255)**	**(0.0488)**	**(00:02.271)**	(0.0193)	**(0.04)**	(00:00.339)	**(0.0179)**	(0.0428)	**(00:00.401)**
C	0.3512	0.8471	00:45.160	**0.2622**	**0.7381**	**00:33.351**	0.2629	0.4205	00:45.694	**0.2304**	**0.4297**	**00:31.953**
	(0.024)	(0.0479)	(00:00.830)	**(0.0249)**	**(0.0584)**	**(00:01.130)**	(0.0163)	(0.0341)	(00:00.529)	**(0.021)**	**(0.0387)**	**(00:00.505)**
D	0.3985	0.8714	01:01.851	**0.3351**	**0.844**	**00:46.192**	0.2415	0.4439	01:03.647	**0.2283**	**0.4314**	**00:43.879**
	(0.0209)	(0.0494)	(00:01.004)	**(0.0257)**	**(0.045)**	**(00:01.139)**	(0.0131)	(0.0278)	(00:01.143)	**(0.0126)**	**(0.037)**	**(00:00.487)**
E	0.4778	0.9129	01:34.769	**0.4672**	**0.9028**	**01:08.612**	**0.3073**	**0.3698**	01:34.197	0.3277	0.3819	**01:06.508**
	(0.0207)	(0.0303)	(00:01.455)	**(0.0226)**	**(0.0324)**	**(00:01.396)**	**(0.0151)**	**(0.047)**	(00:01.501)	(0.0178)	(0.0574)	**(00:01.770)**

TABLE III: RQ2: Results using $complete$ and $random$ initialization, times denoted as mm:ss:x.

Init.	$complete$ initialization						$random$ initialization					
Results	Baseline (fixed)			Contribution (fixedXORgen)			Baseline (fixed)			Contribution (fixedXORgen)		
Input	NRP		Time	NRP		Time	NRP		Time	NRP		Time
model	best	median	median	best	median	median	best	median	median	best	median	median
A	**0.457**	0.446	00:10.0	**0.457**	**0.457**	**00:08.2**	0.454	0.439	00:11.4	**0.461**	**0.440**	**00:08.6**
B	0.526	0.504	**00:35.3**	**0.582**	**0.556**	00:40.7	0.589	**0.554**	00:38.4	**0.602**	0.540	**00:30.0**
C	0.379	0.357	00:47.8	**0.459**	**0.435**	**00:43.3**	0.508	0.461	00:59.2	**0.539**	**0.491**	**00:46.4**
D	0.314	0.276	**01:04.3**	**0.427**	**0.405**	01:35.8	0.434	0.403	01:20.9	**0.473**	**0.443**	**01:06.8**
E	0.218	0.207	**01:48.6**	**0.357**	**0.336**	02:25.1	0.272	0.226	02:15.0	**0.330**	**0.290**	**01:39.1**

Summary of results (RQ2). We find that even for a single-objective version of the NRP problem, supporting user-provided domain knowledge (in the sense of a manually specified initial operator) can lead to an improvement: The combined operator based on our generated rules and the manually specified operator produces better solutions than considering only the manually specified one, sometimes while at the same time improving the execution time.

D. Limitations and Threats to Validity.

The main limitation of our technique is that generating the mutation operator using the upper tier takes time and is required before execution of the lower tier that actually finds solutions. The impact of this overhead depends on how often the lower tier is ran using the generated mutation operator.

External validity is threatened by the limited scope of our experiments: one benchmark case with five models of varying size, including some of significant size. Regarding conclusion validity, our analysis is restricted to descriptive statistics instead of a more rigorous inferential methodology. This treatment is justified by the nature of our data, especially by the observed standard deviations, which are generally much smaller than the differences of means—indicating a clear difference between the treatments. Construct validity is threatened because we only consider two quality indicators (albeit particularly common ones), whereas other quality indicators could lead to different results [29].

V. Related Work

Hyper-heuristics. Hyper-heuristics can be described as "off-the-peg" methods that, given a particular problem class and a set of low-level heuristics components, can automatically produce an adequate combination of these components to effectively solve a problem instance [30]. Hyper-heuristics are categorized by their objective (*selection* or *generation* of heuristics) and source of feedback (*online* or *offline* learning) [31]. Online hyper-heuristics learn while solving a problem instance, while offline approaches gather knowledge from a training set that can hopefully generalise to unseen problem instances. In this paper, we present an offline hyper-heuristic to *generate* efficient mutation operators for multi-objective search-based MDE optimization problems.

Automated generation of mutation operators for vector-based encoding. Self-adaptive mutation [32] has been proposed as a means to change the mutation rate of genetic algorithms using vector-based encoding. Meta-learning has been used to design mutation operators expressed as register machines [33] and in genetic programming [34]. Woodward and Swan [33] show how register machines can be used to express two common mutation operators (i.e., one-point and uniform mutation). Hong et al. [34] use genetic programming to automatically generate mutation operators over a benchmark set that can outperform existing human-designed mutation operators. These works were focused on single-objective problems. Recently, focusing on search-based refactoring, Abid et al. [35] show how association rule mining can seed the initial population and derive smart change operators. Their approach is specifically tailored to refactoring.

Automated generation of mutation operators in MDE. On generating mutation operators for model-driven optimization, there is a technique by Burdusel et al. [3], [36]. Burdusel et al. have a different goal than us, as they aim to establish atomicity (and not efficiency) of the generated mutation operators. Atomic mutation rules, such as the manually specified rules for the NRP case considered in our evaluation (see Fig. 2), encapsulate atomic changes to a solution. Hence, this technique does not produce operators such as those observed in our evaluation (e.g., assigning several artifacts at the same time), since they are not atomic. Addressing a broader context than optimization, Gómez-Abajo et al. [37] present Wodel, a domain-specific language, and Eclipse plugin to develop and analyse domain-independent model-based mutation operators. While Wodel can aid in the creation of mutation operators, it still requires the user to define the mutation rules [37]. Also related to the generation of mutation operators, there are research studies on analysing generated mutation operators. Kosiol et al. [38] present an approach to check whether mutation rules improve solution consistency or at least do not introduce new violations.

VI. Conclusions and Future Work

We present an approach for automatically generating efficient mutation operators for multi-objective optimization problems. We extend the FitnessStudio technique and its two-tier framework with a novel component for evaluating problem-specific mutation operators for multi-objective optimization problems. We provide means to users provide domain knowledge (i.e., an initial mutation operator) and set custom configurations to fine-tune the upper-tier's framework. These principles make our approach suitable to address other available multi-objective software engineering problems. Based on our evaluation, we found evidences that our approach can achieve an improved performance without sacrificing result quality compared to manually-specified operators.

As future work, we foresee three directions. First, the timeout parameter when generating mutation operators might affect the efficiency of the generated mutation operator. Thus, increasing the timeout further might increase efficiency of our technique. Second, as our research showed, the used initialization method can have a significant influence on results both when generating and applying mutation operators. Then, we believe that the initialization methods should be investigated more deeply, especially for finding cases when our technique should be used. Other evaluation metrics, such as CPU time and memory usage, may be helpful in this analysis.

References

[1] M. Harman and B. F. Jones, "Search-based software engineering," *Information and software Technology*, vol. 43, 2001.

[2] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, 1994.

[3] A. Burdusel and S. Zschaler, "Towards automatic generation of evolution rules for model-driven optimisation," in *Software Technologies: Applications and Foundations*, 2018.

[4] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical software engineering and verification*, 2010.

[5] C. Henard, M. Papadakis, M. Harman, and Y. Le Traon, "Combining multi-objective search and constraint solving for configuring large software product lines," in *ICSE*, vol. 1, 2015.

[6] J. M. Horcas, D. Strüber, A. Burdusel, J. Martinez, and S. Zschaler, "We're not gonna break it! consistency-preserving operators for efficient product line configuration," *TSE*, 2022.

[7] M. Bowman, L. C. Briand, and Y. Labiche, "Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms," *TSE*, vol. 36, 2010.

[8] D. Strüber, "Generating efficient mutation operators for search-based model-driven engineering," in *ICMT*, 2017.

[9] M. Fleck, J. Troya Castilla, and M. Wimmer, "The class responsibility assignment case," *TTC*, 2016.

[10] A. J. Bagnall, V. J. Rayward-Smith, and I. M. Whittley, "The next release problem," *Information and software technology*, vol. 43, 2001.

[11] R. M. Hierons, M. Li, X. Liu, J. A. Parejo, S. Segura, and X. Yao, "Many-objective test suite generation for software product lines," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, 2020.

[12] H. Saada, M. Huchard, C. Nebut, and H. Sahraoui, "Recovering model transformation traces using multi-objective optimization," in *ASE*, 2013.

[13] U. Mansoor, M. Kessentini, P. Langer, M. Wimmer, S. Bechikh, and K. Deb, "MOMM: Multi-objective model merging," *Journal of Systems and Software*, vol. 103, 2015.

[14] H. Abdeen, D. Varró, H. Sahraoui, A. S. Nagy, C. Debreceni, A. Hegedüs, and A. Horváth, "Multi-objective optimization in rule-based design space exploration," in *ASE*, 2014.

[15] M. T. Jensen, "Helper-objectives: Using multi-objective evolutionary algorithms for single-objective optimisation," *Journal of Mathematical Modelling and Algorithms*, vol. 3, 2004.

[16] Y. Zhang, M. Harman, and S. A. Mansouri, "The multi-objective next release problem," in *GECCO*, 2007.

[17] S. John, A. Burdusel, R. Bill, D. Struber, G. Taentzer, S. Zschaler, and M. Wimmer, "Searching for optimal models: Comparing two encoding approaches," in *ICMT*, 2019.

[18] T. Stahl, M. Völter, and K. Czarnecki, *Model-driven software development: technology, engineering, management*. Wiley, 2006.

[19] A. Burdusel, S. Zschaler, and D. Strüber, "MDEOptimiser: A search based model engineering tool," in *MoDELS*, 2018.

[20] A. Konak, D. W. Coit, and A. E. Smith, "Multi-objective optimization using genetic algorithms: A tutorial," *Reliability engineering & system safety*, vol. 91, 2006.

[21] M. Li and X. Yao, "Quality evaluation of solution sets in multiobjective optimisation: A survey," *ACM Computing Surveys*, vol. 52, 2019.

[22] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: advanced concepts and tools for in-place EMF model transformations," in *MoDELS*, 2010.

[23] D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, and M. Tichy, "Henshin: A usability-focused framework for EMF model transformation development," in *ICGT*, 2017.

[24] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[25] J. J. Durillo and A. J. Nebro, "jmetal: A java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, 2011.

[26] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II," in *PPSN*, 2000.

[27] The authors, "Online Appendix," 6 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6645808

[28] T. G. N. Da Silva, L. S. Rocha, and J. E. B. Maia, "An effective method for mogas initialization to solve the multi-objective next release problem," in *MICAI*, 2014.

[29] M. Ravber, M. Mernik, and M. Črepinšek, "The impact of quality indicators on the rating of multi-objective evolutionary algorithms," *Applied Soft Computing*, vol. 55, pp. 265–275, 2017.

[30] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu, "Hyper-heuristics: a survey of the state of the art," *Journal of the Operational Research Society*, vol. 64, 2013.

[31] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward, "A Classification of Hyper-heuristic Approaches," in *Handbook of Metaheuristics*, Boston, MA, 2010.

[32] D. Smullen, J. Gillett, J. Heron, and S. Rahnamayan, "Genetic algorithm with self-adaptive mutation controlled by chromosome similarity," in *CEC*, 2014.

[33] J. R. Woodward and J. Swan, "The automatic generation of mutation operators for genetic algorithms," in *GECCO*, 2012.

[34] L. Hong, J. H. Drake, J. R. Woodward, and E. Özcan, "A hyper-heuristic approach to automated generation of mutation operators for evolutionary programming," *Applied Soft Computing*, vol. 62, 2018.

[35] C. Abid, D. Rzig, T. Ferreira, M. Kessentini, and T. Sharma, "X-SBR: On the Use of the History of Refactorings for Explainable Search-Based Refactoring and Intelligent Change Operators," *TSE*, 2021.

[36] A. Burdusel, S. Zschaler, and S. John, "Automatic generation of atomic multiplicity-preserving search operators for search-based model engineering," *Software and Systems Modeling*, 2021.

[37] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo, "A tool for domain-independent model mutation," *Science of Computer Programming*, vol. 163, 2018.

[38] J. Kosiol, D. Strüber, G. Taentzer, and S. Zschaler, "Sustaining and improving graduated graph consistency: A static analysis of graph transformations," *SCP*, vol. 214, 2022.