Noname manuscript No.
(will be inserted by the editor)

Automatic Generation of Atomic Multiplicity Preserving Search Operators for Search-Based Model Engineering

Alexandru Burdusel · Steffen Zschaler · Stefan John

Received: date / Accepted: date

Abstract Recently there has been increased interest in combining Model-Driven Engineering (MDE) and Search-Based Software Engineering (SBSE). Such approaches use meta-heuristic search guided by search operators (model mutators and sometimes breeders) implemented as model transformations. The design of these operators can substantially impact the effectiveness and efficiency of the meta-heuristic search. Currently, designing search operators is left to the person specifying the optimisation problem. However, developing consistent and efficient search-operator rules requires not only domain expertise but also in-depth knowledge about optimisation, which makes the use of model-based meta-heuristic search challenging and expensive. In this paper, we propose a generalised approach to automatically generate atomic multiplicity preserving search operators (aMPSOs) for a given optimisation problem. This reduces the effort required to specify an optimisation problem and shields optimisation users from the complexity of implementing efficient meta-heuristic search mutation operators. We evaluate our approach with a set of case studies, and show that the automatically generated rules are comparable to, and in some cases better than, manually created rules at guiding evolutionary search towards near-optimal solutions.

Keywords model driven optimisation · search-based software engineering · multi-objective optimisation

Alexandru Burdusel
Department of Informatics
King's College London
30 Aldwych, London, WC2B 4BG
E-mail: alexandru.burdusel@kcl.ac.uk

Steffen Zschaler
Department of Informatics
King's College London
30 Aldwych, London, WC2B 4BG
E-mail: szschaler@acm.org

Stefan John
Department of Informatics
Philipps-Universität Marburg
Hans-Meerwein-Straße 6, Marburg, 35043
E-mail: stefan.john@uni-marburg.de

1 Introduction

Search-based software engineering (SBSE) [23] has seen increasing interest over the past decade. SBSE views software engineering as a problem of searching a, potentially very large, design space for optimal solutions and proposes techniques and tools for automating this search, frequently using meta-heuristic search techniques. As a result, more design alternatives can be explored more quickly than would be possible manually. More recently, there has been an increasing interest in applying SBSE techniques in the context of MDE [9], making the benefits of domain-specific modelling languages (DSMLs) available in an SBSE context.

Typical approaches (*e.g.,* [2, 18]) use evolutionary algorithms (EA). Users provide small endogenous model transformations (*e.g.,* expressed as Henshin rules [43]) to specify mutation operators, which are then used for generating new candidate solution models. Writing these transformations is difficult: naïve implementations can easily cause the search to get stuck in local optima or to work very inefficiently.

In this paper, we present a novel technique for automatically generating mutation operators from a declarative specification of an optimisation problem. We generate operators that are consistency preserving, a key property for enabling the search to move out of local optima (without consistency preservation, operators would temporarily try to invalidate constraints, which would be penalised by the search algorithm). In particular, the approach we describe is focused on multiplicity constraints. We call such operators *multiplicity-preserving search operators* (MPSOs).

We will show, through case-study–based experimental evaluation, that our automatically generated MPSOs result in search that is at least as efficient and effective as (and in some cases better than) search based on rules created manually. At the same time, the automatic generation avoids the complexity and effort of manual creation and reduces the likelihood of erroneous or sub-optimal search operators being used. To the best of our knowledge, only [42] proposed an alternative approach for automatic generation of search operators, based on meta-learning. In contrast, our proposed technique avoids the need for a learning phase for each new problem.

This paper extends the work in [13], where we provided a general description and classification of MPSOs, followed by a description and evaluation of an algorithm for generating atomic MPSOs that preserve multiplicity constraints. In this paper, we add the following additional material:

1. Detailed examples of rule application patterns for the generated aMPSOs;
2. A description and examples of the types of generated iterative aMPSOs;
3. Additional experiments to evaluate the efficiency of aMPSOs, in particular with respect to non-multiplicity constraints;
4. An analysis of the impact of changing mutation step size when using aMPSOs; and
5. A more extensive discussion of related work.

The remainder of this paper is structured as follows: In Sect. 2 we introduce some relevant background, followed by a running example in Sect. 3. Section 4 contains the main contributions, describing MPSOs and the generation algorithm. Section 5 presents the experimental setup, followed by Sect. 6 in which we discuss results. In Sect. 7 we evaluate related work.

2 Background

In this section, we briefly describe the relevant background to our research. In particular, we cover key MDE concepts, followed by an introduction to Search-Based Model Engineering (SBME) and a discussion of higher-order transformations.

Model-driven engineering MDE considers models to be the primary artefact in software development [7]. Models are expressed in higher-level languages providing abstractions that are just right for the problem to be solved. Such languages are often called domain-specific modelling languages (DSMLs), and their (abstract) syntax is captured in metamodels (object-oriented models of the language concepts and their relationships). Model transformations—programs that take one or more models and produce new model(s) from them—are fundamental to MDE and to the powerful automation support it provides. Model transformations are often expressed using specialised languages and tools. Henshin [43] is one example, based on graph-transformation theory.

Search-based model engineering Search-based approaches in software engineering often use evolutionary search techniques. Evolutionary search (ES) [17] starts from a population of candidate solutions and evolves these iteratively by applying mutation (and possibly breeding) operators to generate new candidate solutions. In each evolution step, all new candidate solutions' fitness is evaluated against the provided objective functions and this is used to rank solutions and select the best ones to carry over to the next generation. This process is repeated until a given number of iterations is reached or a different stopping condition is met. A particular type of evolutionary algorithms are multi-objective evolutionary algorithms (MOEAs) [17], which can handle multiple, possibly conflicting objective functions. A common problem with ES is that it may get stuck in so-called local optima; that is, solutions that are better than their neighbours (solutions that can be reached by a single application of a mutation operator) but that are not globally optimal. A typical reason for algorithms to get stuck in a local optimum is the inability of the mutation operators to generate solutions that are better than the current best solution found . In this paper, we aim to generate atomic mutation operators that seek to alleviate this problem, ensuring that they can always be applied successfully to generate new search solution candidates.

Evolutionary algorithms have been applied to MDE in multiple ways [9,26]: some approaches (*e.g.,* [2,18]) encode candidate solutions as sequences of transformation rules applications and apply genetic algorithms to solve the search problems. Other approaches (*e.g.,* [47]) directly use models as candidate solutions. In both cases, model transformations are used to specify the available mutation operators. Fitness functions and constraints are specified as model queries using OCL or Java.

Higher-order transformations The term higher-order transformations (HOTs) [44] refers to transformations that produce new model transformations. These are particularly useful when building advanced tools for MDE. In this paper, we are building on work on HOTs in two areas: generating consistency-preserving edit operations and generating model-repair transformations.

In [28], the authors introduce the SiDiff Edit Rule Generator (SERGe). SERGe is an Eclipse plugin to automatically generate consistency preserving edit operations (CPEOs),

encoded as Henshin transformation rules, from an EMF metamodel. A CPEO is an atomic operation that, when applied to a consistent model instance, always generates a transformed consistent model instance. SERGe generates a complete set of CPEOs that can generate or delete any consistent model instance through repeated applications. SERGe requires input metamodels to adhere to additional constraints on the supported multiplicities [27, Sect. 7.3.1]. Our rule-generation algorithm is based on the SERGe algorithm but additionally modifies the generated rules to ensure efficient search.

The term model repair refers to the process of evolving an inconsistent model to make it consistent with its metamodel. In [38], the authors propose an approach for automatically generating repair operators encoded as Henshin rules, which can be used to repair an inconsistent model. The generated repair rules can be applied in a semi-interactive way to transform an invalid model into a valid instance of the metamodel. We make use of the catalogue of repair operations identified in [38].

Henshin Model Transformations In this paper, we use Henshin as a model transformation framework [6]. Henshin is an Eclipse plugin that offers an in-place model transformation language built to run directly on EMF models. Users can specify model transformations using either a diagram editor or an XText based DSL [43]. Henshin uses typed attributed graph theory to encode transformations for EMF models [8].

A Henshin transformation rule consists of a left-hand side (LHS) and a right-hand side (RHS) graph. Henshin transformation rules can be applied both in a deterministic and non-deterministic way, configurable from the transformation engine. Through deterministic rule application, the tool applies the matches found for a transformation rule in sequential order. When using non-deterministic matching, Henshin randomly selects a match to apply from the list of matches found.

The LHS of a Henshin transformation rule supports specifying application conditions to identify the conditions under which a transformation rule can be applied. Application conditions are patterns used to indicate the absence or presence of a graph pattern in a model [6]. A negative application condition (NAC) specifies a graph pattern to indicate the absence of a subgraph before the graph transformation is applied, while a positive application condition (PAC), specifies a graph pattern to indicate that a subgraph is present before the graph transformation is applied.

The Henshin visual syntax uses colours and tags to highlight the presence of a node or edge in the LHS or RHS of the graph transformation rule. Nodes and edges found in the LHS graph are marked by ≪delete≫ and ≪preserve≫, while nodes found in the RHS graph are marked by ≪create≫ and ≪preserve≫. NACs for nodes and edges are marked with ≪forbid≫ (to indicate that a node should not be present). PACs are marked with ≪require≫ (to indicate that a node or edge should be present). Henshin uses the red color to mark elements with ≪delete≫ tags, grey for ≪preserve≫ elements, green for ≪create≫ elements, blue for ≪forbid≫ elements and brown for ≪require≫ elements.

MDEOptimiser MDEOptimiser (MDEO) [1] is an SBME optimisation tool that allows users to specify optimisation problems in MDE using a DSL. The tool can be used as an Eclipse plugin as well as in standalone mode using a command line interface. With the help of the Scale [11] experiments orchestration language, MDEO can be used to run large scale parallelised experiments using Amazon Web Services cloud infrastructure. The search algorithms supported by MDEO are implemented using MOEAFramework [2].

Figure 1 shows an overview of the inputs required to specify a problem to be solved using MDEOptimiser. The user must provide a set of inputs consisting of a problem description and a problem instance model.

The set of required user inputs is composed of the following elements:

- A problem metamodel describing the structure of problems and solutions;
- A set of endogenous model transformations typed over the problem metamodel, called mutation operators;
- A set of solution constraints. These are either multiplicity constraints refining the problem metamodel multiplicities or additional well-formedness constraints implemented using OCL or Java. These constraints together with the problem metamodel are used to define the solution metamodel, that is, the metamodel to which all valid problem solutions conform to;
- A set of objective functions implemented as OCL or Java queries over solution models;
- A valid instance of the problem metamodel, providing initial problem constraints;

Using these inputs, MDEO runs an ES algorithm to find near-optimal models. The input model is used to generate the initial population by making one copy for each population individual followed by a random mutation to ensure variation. The tool uses the specified mutations to generate new candidate solutions in each algorithm step. Figure 2 shows an overview of how new search solution candidates are generated at each algorithm step. Candidate solutions are evaluated after each generation, using the specified constraint and objective functions. Algorithm 1 shows the pseudocode of an evolutionary algorithm that uses only mutation to generate new search solution candidates. MDEO currently supports mutation only evolutionary algorithms. Ef-

[1] `https://mde-optimiser.github.io`
[2] `https://moeaframework.org`

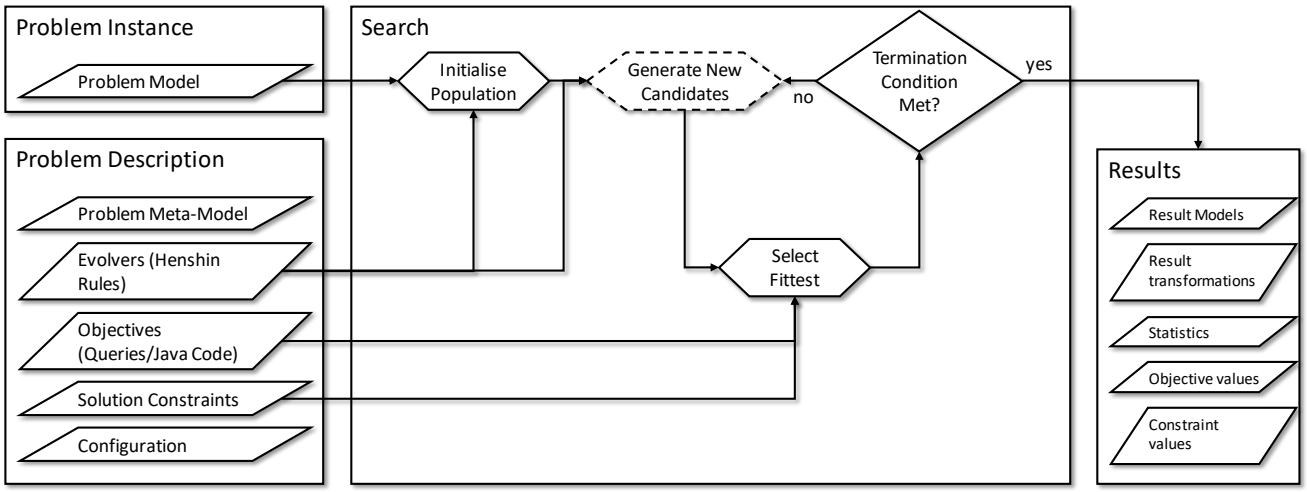


Fig. 1: MDEOptimiser approach general overview [26]

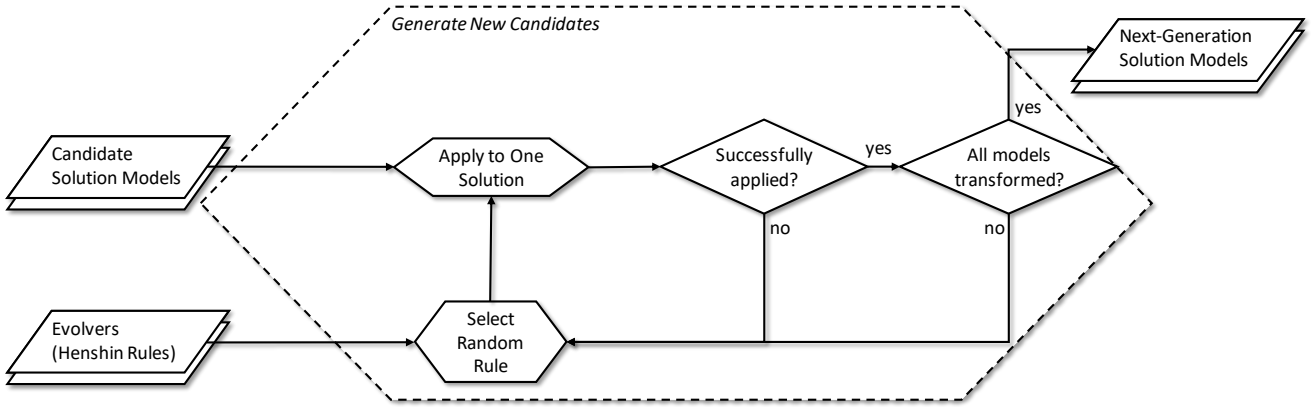


Fig. 2: MDEOptimiser new candidate generation overview [26]

ficient breeding operators for graph structured models are difficult to produce.

Algorithm 1 Abstract Mutation Only EA

```
1:  procedure EA(μ, λ)    ▷ μ evolved solutions generate λ offspring
2:      Population ← []
3:      for μ times do
4:          Population ← Population ∪ INITIALISESOLUTION()
5:      end for
6:      Population ← EVALUATE(Population)
7:      repeat
8:          NewSolutions ← []
9:          while SIZE(NewSolutions) < λ do
10:             C ← SELECT(Population)
11:             NewSolutions ← NewSolutions ∪ MUTATE(COPY(C))
12:         end while
13:         Population ← EVALUATE(Population, NewSolutions)
14:     until Termination Condition
15:     return Population
16: end procedure
```

Evolutionary Search Parameter Control When using EAs, a key challenge arises from the need to configure ideal algorithm parameters, both, off-line, before the start of the search and, on-line during the search [3]. The parameters used at the start and during execution can help steer the search towards having a greater chance of finding near-optimal solutions. Common evolutionary algorithm parameters that can have a direct impact on algorithm performance and the quality of produced solutions are: *crossover rate*, used to control the probability of applying the recombination operators when generating a new offspring solution; *mutation rate* which controls the probability of applying a mutation to a new solutions; *mutation step size* denotes the degree of changes caused by a mutation operator to an offspring solution; *population size* controls the size of the algorithm population; and *termination condition* defines the search algorithm stopping criteria (*e.g.,* a certain number of algorithm steps have elapsed or there is no significant solution quality improvement) [17].

The EA parameter search process can be separated in two phases based on the stage when it takes place [17]:

– *Parameter tuning* is used before starting the search and the aim is to find the ideal parameter values to start the EA with;
– *Parameter control* is concerned with changing parameter values during search.

3 Running Example

In this section, we introduce a running example of an SBME optimisation problem that can be specified using MDEO. Consider the scenario of a software development team who use Scrum as an agile software development methodology. Scrum is a process management framework that proposes the use of fixed time iterations, also called sprints, during which a set of tasks defined as user stories are implemented, tested and released into the product under development [39].

We will briefly introduce the core Scrum concepts as described in [39]. The key artifacts of Scrum are the product, the product backlog and the sprint backlog. The product backlog is the list of all user stories that, when implemented, will result in a completed product. The sprint backlog is the list of user stories which the team aims to complete in a sprint. Each user story has associated story points, which serve as an estimate of the effort needed to complete it. The product owner is in charge of prioritising the backlog to make sure the most important user stories are worked on first. For the duration of a project, the development team completes several sprints. The average number of story points resulting from the completed user stories in a sprint is also known as team velocity.

In our example, we will consider that the user stories forming the backlog have an `Importance` metric, denoting how important they are for a stakeholder, in addition to the `Effort` metric, which shows the required effort for completion. The product owner is required to prioritise these tasks so that the average stakeholder importance is equally distributed across the sprints required to implement the work items in the backlog. We call this objective the Stakeholder Satisfaction Index, and we calculate it as the standard deviation of average stakeholder importance across sprints.

In Fig. 3 we show a metamodel of this problem. The goal of the problem is to assign `WorkItem` elements to a number of `Sprints` with the following objectives: **Objective 1** minimise the `Sprint` effort deviation; **Objective 2** minimise the Stakeholder Satisfaction Index. In Listings 1 and 2 the two problem objective functions are given in OCL. Objective 1 calculates the standard deviation of `Sprint Effort`. This objective is minimised to ensure that all `Sprints` have close to identical `Effort` values. Objective 2 first calculates the `Importance` standard deviation for each `Stakeholder` across all `Sprints`. Then, to ensure that all stakeholders have an even `Importance` distribution across all `Sprints`, the objective calculates the standard deviation of all `Stakeholders` `Importance` distributions. This objective is minimised to prefer solutions with small standard deviations between *Stakeholders* `Importance` distributions.

```
context Plan def: MinSprintEffortDeviation : Real =
  self.model.sprints->collect(committedItem.effort->sum(
      )).standardDeviation()
```

Listing 1: SP Objective 1 in OCL

```
context Plan def: CustomerSatisfactionIndex : Real =
  self.stakeholders
    ->collect(sh |
      sh.sprints
        ->collect(
            committedItem
              ->select(ci | ci.stakeholder = sh)
              .importance
              ->sum())
        .standardDeviation())
    .standardDeviation()
```

Listing 2: SP Objective 2 in OCL

The problem has the following constraints: **Constraint 1** all `WorkItem` entities must be assigned to a `Sprint`; **Constraint 2** no solution must have fewer `Sprints` than total backlog effort divided by team velocity. In Listings 3 and 4 we give the two problem constraints in OCL. Constraint 1 counts the number of `WorkItem` entities that are not asigned to a `Sprint`. This constraint is equivalent with refining the metamodel multiplicity from a lower bound of 0 to a lower bound of 1 for the `sprints` edge between a `Plan` and `Sprint` and also for the `isPlannedFor` edge between a `WorkItem` and a `Sprint`. Constraint 2 calculates the total number of `Sprints` desired using total `WorkItems` `Effort` and the maximum `Effort` that can be delivered in a `Sprint` and the ensures that there are no planned `Sprints` with more `Effort` than the team `Effort` velocity.

To explore the search space of the Scrum Planning problem, the mutation operators must create `Sprint` entities and assign `WorkItem` elements to them, until all the `WorkItem` elements belong to a `Sprint`. In Fig. 4 we include the mutation operators implemented manually for this case study. Fig. 4a shows the mutation operator to create a new `Sprint` and assign to it a `WorkItem` that has not already been assigned to another `Sprint`. In Fig. 4b we include the operator that deletes an empty `Sprint`, which deletes a `Sprint` that has no `WorkItems` assigned to it. In Fig. 4c we include the operator used to add `WorkItems` to an existing `Sprint`, ensuring that any `WorkItems` allocated by this operator are not already assigned to another `Sprint`. Finally, in Fig. 4d we include a mutation operator that unassigns a `WorkItem` from a `Sprint` and assigns it to another `Sprint`. This operator can create empty `Sprints`, which can then be deleted by the delete sprint operator in Fig. 4b.

Readers familiar with constraint solving may be tempted to argue that this specific problem could be solved using an

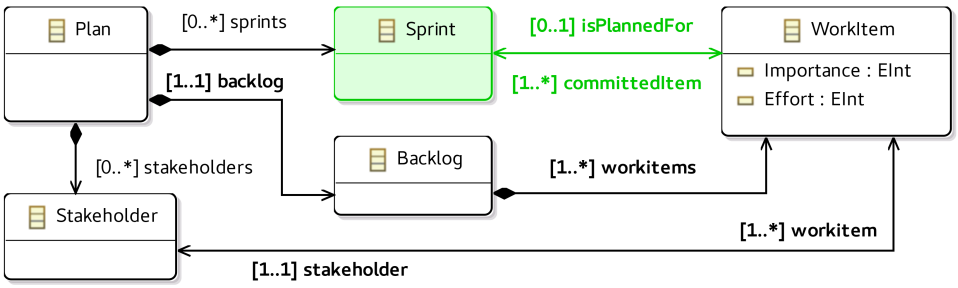
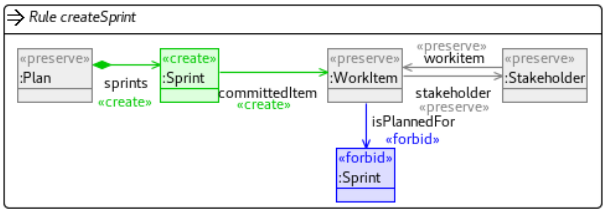
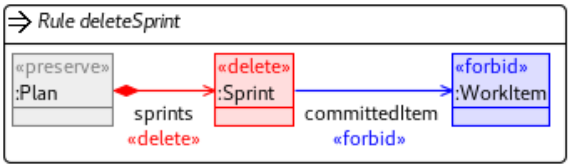


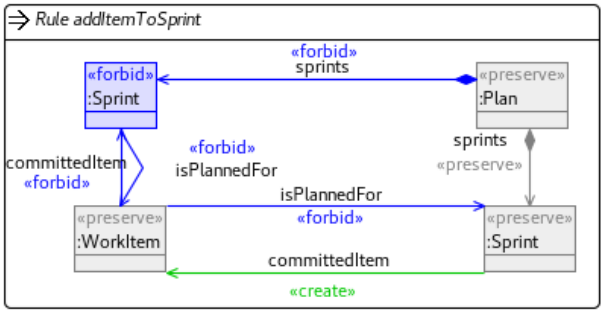
Fig. 3: Scrum Planning metamodel.



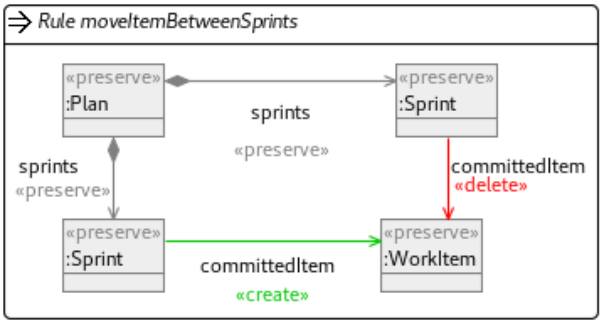
(a) Create Sprint



(b) Delete Sprint



(c) Add WorkItem to Sprint



(d) Move WorkItem between Sprints

Fig. 4: Summary of the mutation operators implemented manually for the Sprint Planning case study.

```
1  context Plan def: NoUnassignedWorkItems : Integer =
2    self.backlog.workitems
3    ->select(isPlannedFor->isEmpty())->size()
```

Listing 3: SP Constraint 1 in OCL

```
1  context Plan def: AllowedMaxSprints : Integer =
2    let effort = self.backlog.workitems.effort->sum() in
3    let maximumVelocity = 25 in
4      let desiredSprints =
5        (effort / maximumVelocity).round() in
6        let nonEmtpySprints =
7          plan.sprints
8            ->select(committedItem->notEmpty)
9            ->size() in
10         if (nonEmptySprints > desiredSprints) then
11           desiredSprints - nonEmptySprints
12         else
13           0
14         endif
```

Listing 4: SP Constraint 2 in OCL

optimising constraint solver like Choco [1]. However, solving the problem in such a way would require substantial encoding effort to express the problem in a format that constraint solvers can understand, which is fairly far away from the original problem description, even when using so-called high-level languages like Essence [21]. Therefore, our work aims to reduce the amount of encoding effort required as much as possible.

4 Generating Mutation Operators

Rather than asking the user to manually specify the mutation operators, our goal is to automatically generate them. In this section, we identify requirements for good mutation operators, introduce a general structure for mutation operators satisfying those requirements, and propose a systematic algorithm for generating them.

As a result, a user will no longer be required to explicitly provide mutation operators as part of the optimisation problem specification. Instead, they will specify the sub-metamodel for which mutation operators should be generated. This explicitly separates the parts of the metamodel

that specify problem constraints from those which hold solution information. In our running example, the user would specify that the Sprint node and all its edges can be modified. This will produce rules that create new Sprints and assign `WorkItems` to them.

4.1 Requirements on mutation operators

Generally, any transformation typed over the problem metamodel could be used as a mutation operator. However, here we are focusing on transformations that make small-granular changes (*e.g.,* adding a node). This will allow a detailed exploration of the search space. To identify additional requirements on mutation operators, we will explore two problems that can occur when operators are constructed naïvely: getting stuck in local optima and changing applicability of rules during different search phases.

The search process can get stuck in *local optima* when the constraints prevent the mutation operators from generating new and diverse individuals with a single transformation application. Consider the Scrum planning use case including the following two operators: one for creating a new `Sprint` and one for moving a `WorkItem` from one `Sprint` to another. Once all the `WorkItem` elements have been assigned to a `Sprint`, no more new `Sprint` nodes can be created: because there are no more free `WorkItem` elements, the lower-bound constraint that no Sprint should be empty can no longer be satisfied for these new `Sprints`. If all the `WorkItems` have initially been assigned to a small number of `Sprints`, and no new `Sprints` can be created, the search will be unable to find solutions that have a good average distribution of `WorkItems` across the created `Sprints`. Note that creating two mutation operators, one to create an empty `Sprint` and one to move an existing `WorkItem` to the newly created `Sprint`, won't solve this problem: until the constraint is satisfied, the search algorithm would have to include the invalid solution in the archive and then apply the required repair operator in one of the following iterations. However, if all the other population individuals are valid, they will dominate the one with the invalid `Sprint`, which will be removed from the population. Generally, this problem is encountered where there are non-zero lower-bound multiplicities. In these cases, we require mutation operators to apply both edit and repair in one step.

The search can be split into *two phases:* in the first phase, all candidate solutions conform to the problem metamodel, but may not yet satisfy the additional solution constraints; in the second phase, all candidate solutions satisfy the additional solution constraints. These two phases potentially require different repair steps. Consider again a mutation operator creating a new Sprint node. In the first phase, the appropriate repair is to find a `WorkItem` that has not yet been assigned to another `Sprint` and assign it to the new `Sprint`.

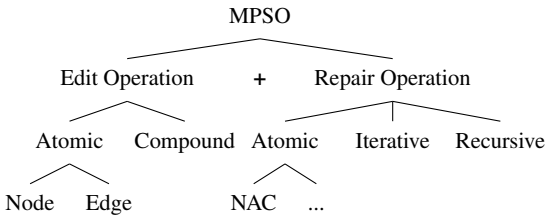


Fig. 5: MPSOs structure

In the second phase, this rule is not applicable anymore because no unassigned `WorkItems` remain. However, there is an alternative repair that takes a `WorkItem` from an existing Sprint with at least two `WorkItem` elements assigned to it. We need to generate appropriate mutation operators for each phase of the search.

Mutation operators that satisfy the requirements imposed by the conformance to the solution metamodel and the additional problem multiplicity constraints, we will call Multiplicity-Preserving Search Operators (MPSOs).

4.2 General structure of MPSOs

As we have seen in the previous sub-section, MPSOs are transformation rules that combine a change to the model (an edit operation) with the necessary repair. In Fig. 5 we show the structure of MPSOs as well as further categorising edit and repair operations. We consider that edit operations can be either atomic or compound (a composition of multiple atomic operators). Atomic operators will either change a single node or a single edge. A repair operation can be atomic, iterative or recursive. Atomic repairs focus on a single edge and will not create or delete nodes beyond the original edit operation. An iterative repair is a combination of multiple atomic repairs for the same edit, for example, where constraints on multiple edges would be broken by the edit. In contrast, a recursive repair creates or removes nodes as part of the repair, requiring recursive repair steps to be considered. In this paper, we only consider atomic edit operations and atomic or iterative repair. We call the resulting operators atomic multiplicity-preserving search operators (aMPSOs).

4.3 Generation algorithm

In our current approach, we focus only on multiplicity constraints. Supporting arbitrary constraints is not a trivial problem, and it is beyond the scope of this paper to also support such constraints with our generation algorithm. In separate work, Kosiol et al. [32] make the first steps towards support for arbitrary constraints for aMPSOs and introduce a formalisation to reason about the impact of graph transformations on graph constraints.

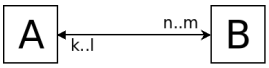


Fig. 6: Multiplicity patterns

In [27, 28], Kehrer et al. introduce the concept of consistency preserving edit operations (CPEOs) and propose a mechanism for automatically generating them from a metamodel with multiplicity constraints. CPEOs can be used as MPSOs in cases where the solution metamodel only has open multiplicities. Any multiplicity is open if the lower bound is zero. Kehrer et al.'s mechanism does not support the generation of CPEOs for edges with closed multiplicities on both sides. Where only one multiplicity is closed, the mechanism only generates a limited range of repairs, which still causes the search to get stuck in local optima.

In this section, we propose an algorithm to generate aMPSOs. We will structure the discussion based on the type of edit operations. For each edit operation, we will then discuss relevant repair actions. We distinguish edit operations for nodes—namely create and delete—and for edges—add, remove, change, and swap. The available repair operations depend on the multiplicity pattern. Fig. 6 shows the labels we will use in our discussion below. In Fig. 7 we include an example metamodel that contains two nodes, A and B, with multiplicity patterns supported by our rule generation algorithm. We use this metamodel to show examples of generated rules in this section.

For each multiplicity pattern we consider, we aim to generate the minimal set of rules that would allow the search to avoid getting stuck in local optima.

4.3.1 Manipulating nodes

In this section, we describe the repair operations required for manipulating nodes. The types of aMPSOs that we generate for this are composed of the atomic rule to create (delete) a node and a repair operation to connect (disconnect) the created (deleted) node to (from) mandatory neighbours (B nodes). The choice of repairs that can be applied is given by the multiplicity pattern between the node being edited and its neighbours. For some repairs, there are many variants in-between, however, we seek to minimise the number of generated rules, so we only generate the rules described.

Creating a node In this section, we introduce the types of repair operations generated for creating a node A from Fig. 6. For each repair, we include the multiplicity patterns for which the generated repair is applicable. We include a summary of the generated rules in Table 1. Figure 9 shows an example for each type of generated node creation aMPSO.

The generated create node aMPSOs described in this section follow a set of principles that seek to satisfy the multiplicity requirements of a node of type A for nodes of type B that have to be assigned. These principles consist of: either assigning existing B nodes and using a NAC to ensure their constraints are not invalidated; relocating existing B nodes from another single node of type A; or by getting each B from a different existing source node A, making sure that the source node lower bound constraint is not invalidated. We will discuss these three options below. There are other options for repairs (*e.g.* picking multiple Bs from the same A, but not all Bs from the same A), however we are not considering these as we are interested in generating the smallest possible set of rules, while still giving the search algorithm different options for leaving a local optimum.

- *NAC repair*: The first type of aMPSO that we generate, is for creating nodes that have a multiplicity pattern with ($n > 0$). For this case, we generate a rule to create a node of type A and connect it to n existing nodes of type B. If ($l < *$), then a negative application condition (NAC) is added for the connected nodes B to ensure that no upper-bound multiplicity invalidations occur (no more than l nodes of type A assigned for each B). Nodes that have an open multiplicity don't need a repair operation.

  Example NAC repair rules Fig. 8a shows an example of this aMPSO for the SP case study, generated for creating a `Sprint` node, that is connected to a note of type `WorkItem`. The rule includes a NAC for the `WorkItem` node which requires that the `WorkItem` node is not already assigned to a `Sprint` node.
  An example of this aMPSO for multiplicity pattern ($n = 1$) and ($l = 5$) is included in Fig. 9e. The generated aMPSO contains l forbid A nodes connected to node B.
- *Single source lower bound repair*: The second type of aMPSO for creating a node is for creating nodes that have a multiplicity pattern with ($n > 0$) and ($l < *$). This pattern means that A must have at least n nodes of type B assigned to it, and node B can have a limited number of nodes of type A assigned to it. We generate a rule to create a node of type A, and connect it to n nodes of type B. Then, the upper-bound for the existing n nodes of type B is repaired by deleting the edges between the required n nodes of type B from a *single* existing node of type A and creating edges between them and the newly created node of type A. A positive application condition (PAC) is generated for the existing source node A to ensure its lower-bound multiplicity (n) is not broken after the node B used in the repair is unassigned. The multiplicity pattern for this aMPSO partially overlaps with the pattern for *NAC repair*, and when this is the case during the generation stage, both operators are generated.

  Example single source lower bound repair rules In Fig. 8b we show an example of this aMPSO for the SP case study, generated for creating a `Sprint` node, when all

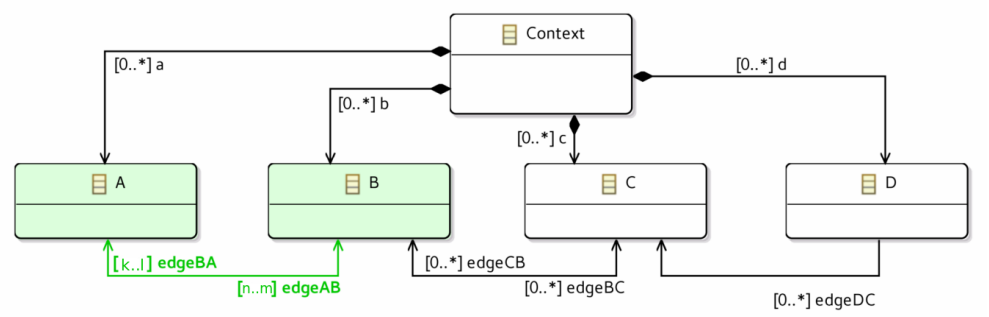


Fig. 7: Metamodel used to show rulegen examples. Nodes A and B have the multiplicity pattern shown in Fig. 6.

Table 1: Create node aMPSOs. In the table, 'c' stands for 'create', 'lb r' for 'lower-bound repair', and 'f#l' for 'forbid l'.

	n=0	**n = 1 and m >n**	**n >1 and m >n**	**n = m**
k ≥ 0 **l >k** **l <***	c A	c A add n B (f#l A) c A lb r single B	c A add n B (f#l A) c A lb r single B c A lb r many B	c A add n B (f#l A)
k ≥ 0 **l = ***		c A add n B		
k = l		c A lb r single B	c A add n B c A lb r single B c A lb r many B	N/A

`WorkItems` are already assigned to other `Sprints`. The rule includes a PAC for the existing `Sprint` node from which the `WorkItem` node used for the repair is taken, to make sure that the lower-bound multiplicity is not invalidated.

An example of this aMPSO for multiplicity pattern ($n = 1$), ($m = *$) and ($k = 1$), ($l = 5$) is included in Fig. 9c.

– *Multiple sources lower bound repair*: The third type of aMPSO for creating a node is for creating nodes that have a multiplicity pattern with ($n > 1$) and ($l < *$). This pattern means that A must have at least n nodes of type B assigned to it, and node B can have a limited number of nodes of type A assigned to it. For this case, we generate a rule to create a node of type A and connect it to n nodes of type B. Then, we repair the upper-bound for the existing n nodes of type B by deleting the edges between the required n nodes of type B from n existing nodes of type A and creating edges between them and the newly created node of type A. A PAC is generated for the existing nodes of type A to ensure that the lower-bound multiplicity is not broken by this operation.

Example multiple sources lower bound repair rule An example of this aMPSO for multiplicity pattern ($n = 2$), ($m = *$) and ($k = 2$), ($l = 2$) is included in Fig. 9d. The rule creates a node A and connects it to 2 mandatory B nodes disconnected from two other A nodes, while ensuring that each source A node still satisfies the lower-bound requirement.

For node pairs that have a fixed multiplicity ($n = m \wedge k = l$), at both ends of any edge, we do not generate a create node aMPSO. Any repair operation for this case requires the creation of the nodes at the opposite end of the edge, and thus a recursive repair.

Deleting a node As with the description for the create operations, we divide the explanation based on repair type. We include a summary of the generated rules in Table 2. Figure 10 shows an example for each type of generated node deletion aMPSO.

Similarly to the generated create node aMPSOs described in the previous section, the delete node aMPSOs described in this section follow a set of principles, that seek to satisfy the multiplicity requirements of a node of type A for nodes of type B, that have to be unassigned. These principles consist of: either unassigning existing B nodes and using a PAC to ensure their constraints are not invalidated; relocating existing B nodes from deleted node A to another single node of type A; or by moving each assigned node B to a different existing node A, making sure that the target node upper bound constraint is not invalidated. We will discuss these three-node deletion repair options below. It is possible to use other repairs (*e.g.,* moving multiple Bs to the same A, but not all Bs to the same A node), however, we are not considering these as we are interested in generating the smallest possible set of rules, while still giving the search algorithm different options for leaving a local optimum.

– *PAC repair*: The first type of aMPSO that we generate is for deleting nodes that have a closed multiplicity ($k > 0$).

Table 2: Delete Node aMPSOs. In the table, 'd' stands for 'delete', 'r lb sg' for 'repair lower bound single', 'r lb mn' for 'repair lower bound multiple', and 'f#m' for 'forbid m'.

	m > n and m < *	**m = ***	**n = m**
k = 0	d A		
k >0 l >k	d A (require each B still has #k A)		
k=l=1	d A r lb sg B (f#m A)	d A r lb sg B	N/A
k=l >1	d A r lb sg B (f#m A) d A r lb mn B (f#m A)	d A r lb sg B d A r lb mn B	N/A

This pattern means that B must have at least k nodes of type A assigned, and each node of type A must be assigned to at least n nodes of type A. For this case, we generate a rule to delete a node of type A and for each of its connected nodes of type B, a PAC is added to ensure that no lower-bound multiplicity invalidations occur after the deletion of the A node. This rule is not generated for cases where $(k = l)$ because nodes with multiplicity $(k = l)$ cannot be repaired with a PAC.

Example PAC repair rules In Fig. 8c we include an example of this aMPSO for the SP case study, generated for deleting a `Sprint` node, that has a `WorkItem` assigned to it. For this example rule, there is no PAC generated for the `WorkItem` because there is no lower-bound multiplicity limit. An example of the generated aMPSO of this type for multiplicity $(k = 2)$ is included in Fig. 10b. The aMPSO deletes node A and requires that node B still has 2 nodes of type B connected to it. For cases when $(k = 0)$, no PAC is generated for node B, and node A is simply deleted. An example aMPSO for this scenario is shown in Fig. 10a.

– *Single target lower bound repair*: This type of aMPSO for deleting a node is for manipulating nodes that have a multiplicity pattern ($k = l$ and $k > 0$). This pattern means that each node of type B must be assigned to k nodes of type A. For this case, we generate a repair to satisfy the lower-bound for the n nodes B, by creating edges between them and another single existing node of type A. A NAC is generated for the existing node A to ensure that the upper-bound multiplicity is not broken if $(m < *)$. This repair ensures that after the A node to which k nodes of type B are assigned is deleted, the B nodes are assigned to another node A not to invalidate their multiplicity constraint.

Example single target lower bound repair rules Fig. 8d shows an example of this aMPSO for the SP case study, generated for deleting a `Sprint` node, that has a `WorkItem` assigned to it. For this example rule, there is no NAC generated because there is no upper-bound multiplicity limit.
Fig. 10c shows an example of this aMPSO for multiplicity pattern $(m = 2)$. Because of the upper-bound multiplicity limit for node A is not $*$, the rule contains a NAC repair. An example of the rule for the case when $(m > 0)$ and no NAC repair is necessary can be seen in Fig. 10d.

– *Multiple target lower bound repair delete*: This type of aMPSO is used for deleting nodes that have a multiplicity pattern with $(k = l)$ and $(l > 1)$. This pattern means that A must have at least n nodes of type B assigned, and each node of type B must be assigned to exactly k nodes of type A. For this case, we generate a repair to satisfy the lower bound for node B by creating edges between them and another existing n nodes of type A. If required, a NAC is generated for the existing nodes of type A to ensure that the upper-bound multiplicity is not broken if $(m < *)$. We only generate this rule for the case where exactly n nodes of type B are attached to the A node to be deleted.

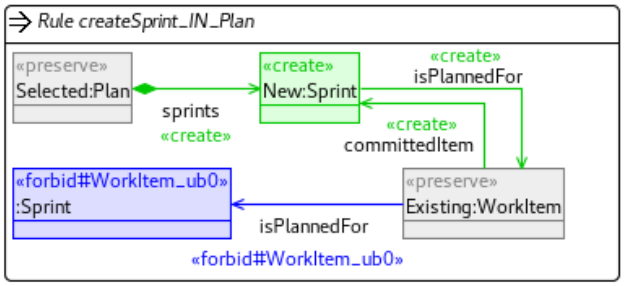
Example multiple target lower bound repair delete rules Fig. 10e shows an example of this aMPSO for multiplicity pattern $(n = 2)$, $(m = 5)$ and $(k = l = 2)$. The rule contains a NAC for each node of type A that is assigned a node of type B from the deleted node B. For the case when no NAC is necessary (*e.g.,* $m = *$), no NAC is generated as seen in the aMPSO shown in Fig. 10f.

For node pairs that have a fixed multiplicity $(n = m \wedge k = l)$, at both ends of any edge, we do not generate a delete node aMPSO. Similar to the create node operations, a repair operation for this case requires the deletion of the node at the opposite end of the node being deleted. We regard this type of operation as recursive, which we will look at in future work.

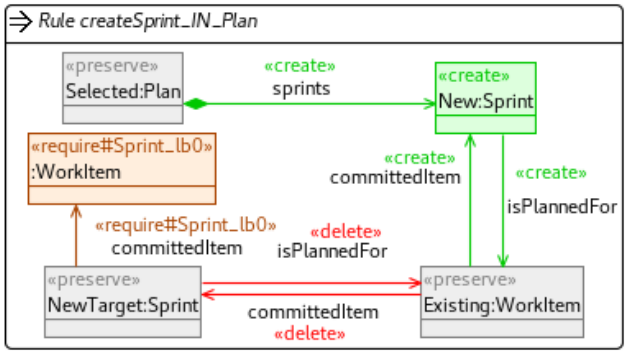
4.3.2 Manipulating edges

In this section, we show the types of aMPSOs we generate for manipulating edges between two nodes. Namely, to add and remove an edge from a node, together with corresponding repair operations. The add and remove edge operations are composed to obtain the more complex change and edge-swap operations. A complete list of the generated edge aMPSOs is included in Tables 3 and 4. Figure 12 shows an example for each type of generated edge creation aMPSO. Figure 14 shows an example for each type of generated edge removal aMPSO.

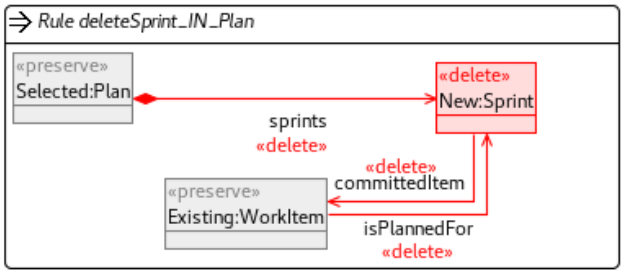
Adding an edge The aMPSO to add an edge between two existing nodes is identical to the add edge CPEO generated by Kehrer et al. This aMPSO includes a NAC to avoid invalidating any upper-bound constraints between the source and target nodes. This aMPSO is generated for all multiplicity patterns except for cases having a fixed multiplicity at one end or at both ends of the connected nodes ($n = m \vee k = l$).



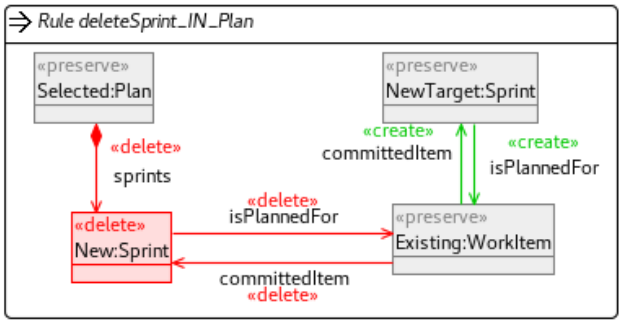
(a) Create Node Rule



(b) Create Node LB Repair Rule



(c) Delete Node Rule



(d) Delete Node LB Repair Rule

Fig. 8: Generated node manipulation aMPSOs for the Scrum Planning case study encoded as Henshin model transformations.

Table 3: Add-edge aMPSOs. In the table (P/N) denotes the presence of optional PAC and NACs that may be required by the source or target node multiplicity.

	m <*	**m = ***	**n=m**
l <*	Add edge NAC A B	Add edge NAC B	Swap edge
l = *	Add edge NAC A	Add edge	Swap edge
k = l	Change edge (P/N A)	Change edge (P/N A)	Swap edge

Table 4: Remove-edge aMPSOs. In the table (P/N) denotes the presence of optional PAC and NACs that may be required by the source or target node multiplicity.

	n = 0	**n >0**	**n = m**
k = 0	Remove edge	Remove edge PAC A	Swap Edge
k >0	Remove edge PAC B	Remove edge PAC AB	Swap Edge
k = l	Change edge (P/N A)	Change edge (P/N A)	Swap Edge

Exmple add edge aMPSOs In Fig. 11a we include an example of an aMPSO for the SP case study that adds an edge between a `Sprint` and a `WorkItem` with a NAC, forbidding that the two nodes are already connected.

Fig. 12d includes an example of the generated aMPSO for this case with multiplicity pattern ($n = 0$), ($m = *$) and ($k = 0$), ($l = *$). The aMPSO has a NAC to avoid being applied in cases when there is already an edge between the two nodes being connected. We include example aMPSOs generated for the case when there is an upper bound ($m = 1$) and ($l = 1$ for both A and B nodes being connected in Fig. 12a. Figures 12b and 12c show example aMPSOs containing NACs for nodes B and A, respectively, when an upper bound limit is present.

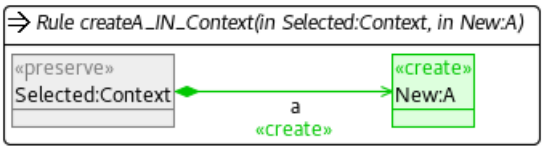
Removing an edge This aMPSO is identical to a CPEO that Kehrer et al. generate, consisting of an operation to remove an edge between two existing nodes A and B.

The generated aMPSO includes a NAC to avoid invalidating any lower-bound constraints between the source and target nodes. This aMPSO is generated for all multiplicity patterns except for cases having a fixed multiplicity at one end or at both ends of the connected nodes ($n = m \vee k = l$).

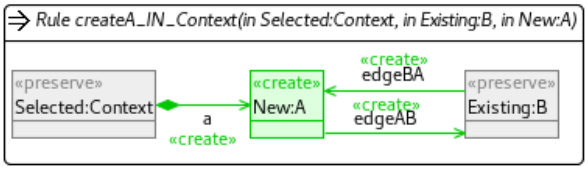
Example remove edge aMPSOs Fig. 14a includes an example of the generated aMPSO for this case with multiplicity pattern $n = 0, m = *and k = 0, l = *$. The generated rules can contain a PAC for node A if there is a lower bound $n = 1$ present (*e.g.,* Fig. 14b), or a PAC for node B if there is a lower bound $k = 1$ present (*e.g.,* Fig. 14c), or a PAC for both nodes A and B if there is a lower bound $n = 1 and k = 1$ present (*e.g.,* Fig. 14d).

In Fig. 11b we include an example of an aMPSO that removes an edge between a `Sprint` and a `WorkItem` with a PAC, requiring that after the application of this rule, the `Sprint` node still has at least one `WorkItem` node still assigned to it, to satisfy the lower-bound multiplicity.

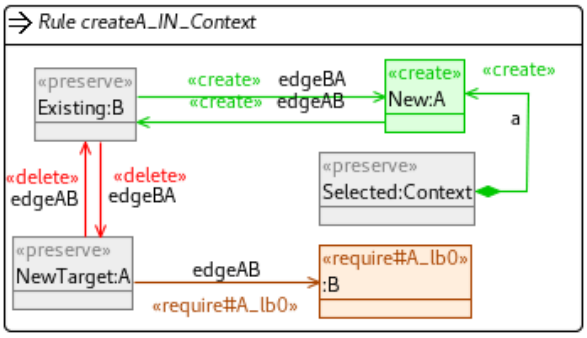
Changing an edge A change edge aMPSO moves a node of type B with a lower bound multiplicity pattern ($k > 0$), to



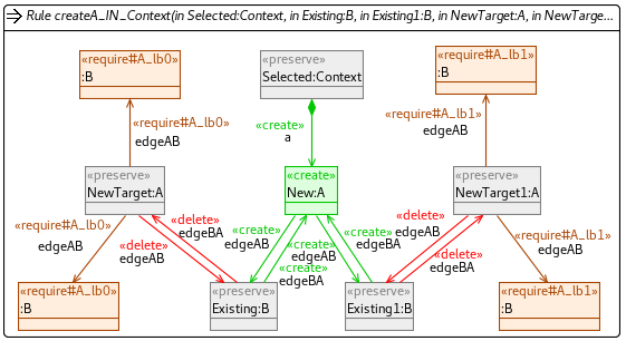
(a) Create node A aMPSO for n = 0, m = * and k = 0, l = 1



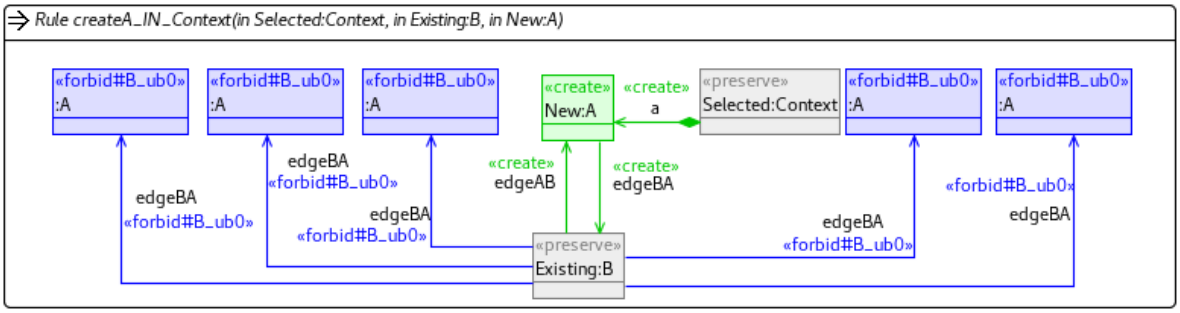
(b) Create node A add n B nodes aMPSO for n = 1, m = * and k = 0, l = *



(c) Create node A lower-bound repair single B aMPSO for n = 1, m = * and k = 1, l = 1



(d) Create node A lower-bound repair many B aMPSO for n = 2, m = * and k = 2, l = 2



(e) Create node A add n B nodes (forbid l A nodes) aMPSO for n = 1, m = * and k = 1, l = 5

Fig. 9: Examples of generated create node aMPSOs

another node of type A, without invalidating the multiplicity constraints. The generated aMPSO includes PAC and NAC conditions to ensure that after the rule application, no lower-bound or upper-bound multiplicities are invalidated for the source and target nodes respectively of type A (to ensure that no node has too many or too few nodes of type B after this rule application). This aMPSO is also generated for closed multiplicity patterns where a multiplicity pattern for either of the connected nodes is fixed (*e.g.*, $n = m \vee k = l$).

Example change edge aMPSOs Fig. 11c shows an example of this aMPSO for the SP case study, generated for changing an edge between a `WorkItem` and two `Sprints`. The rule includes a PAC for the `Sprint` element from which the `WorkItem` element is unassigned to ensure that the lower-bound multiplicity of this node is not invalidated after the application of the rule.

In Fig. 13a we include an example of the generated aMPSO for the generic metamodel in Fig. 7.

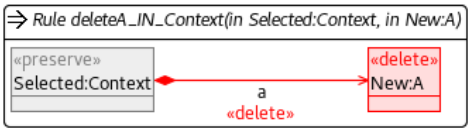
Swapping two edges An edge swap aMPSO is generated for fixed multiplicity patterns on the A side ($n = m$). This operation exchanges two nodes between two pairs of similar node types. For two existing, connected nodes A and B, the aMPSO, finds two other nodes of the same type, A' and B' and disconnects node A from node B and A' from B', and connects node A to B' and A' to B.

Example swap edge aMPSO We include an example of this aMPSO in Fig. 13b, in which two B nodes are exchanged between two A nodes.

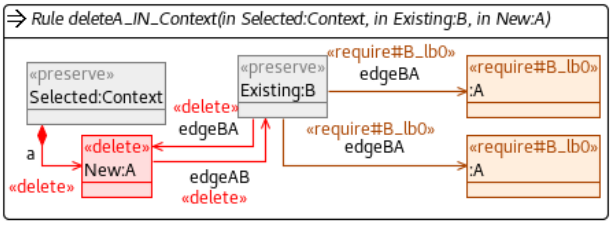
4.3.3 Iterative repair

Iterative repair rules are generated by creating combinations of the possible repair types described above for all the edges of a node that has to be mutated. This approach increases the number of rules generated for nodes that have multiple edges.

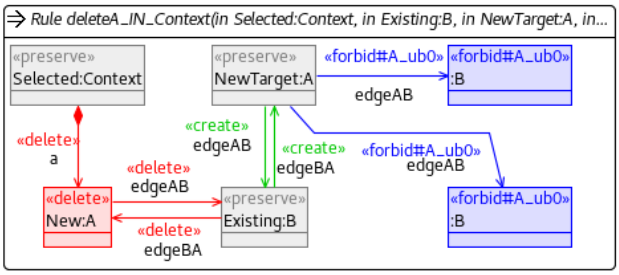
To demonstrate this feature, we include in Fig. 15 a refined version of the metamodel shown in Fig. 7 with changed



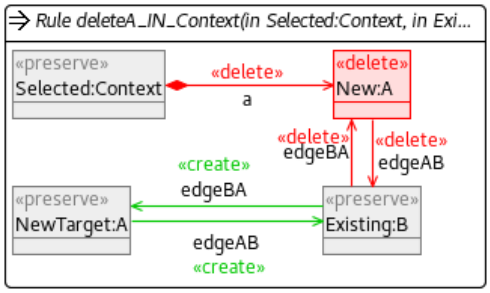
(a) Delete node A aMPSO for m > n, m < * and k = 0



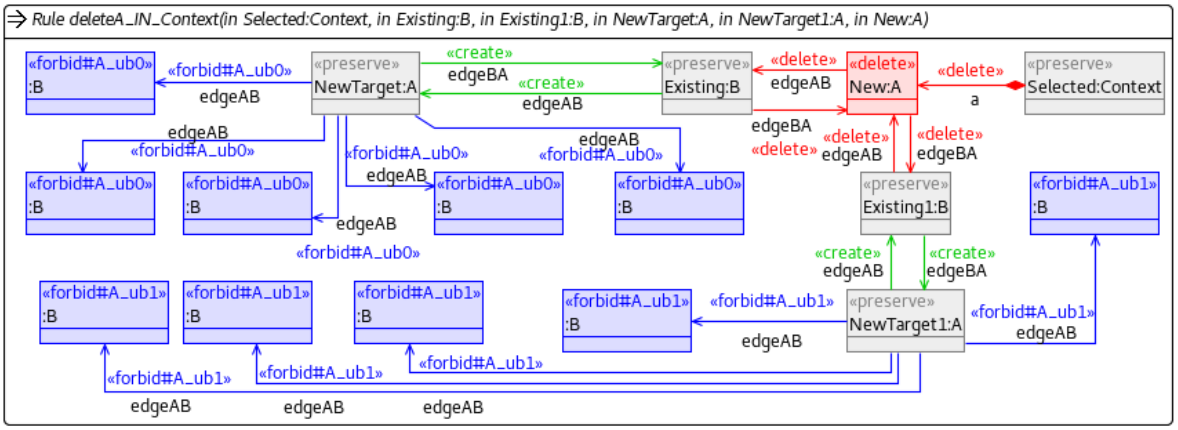
(b) Delete node A require each B still has k A nodes aMPSO for m > n, m < * and k > 0, l > k



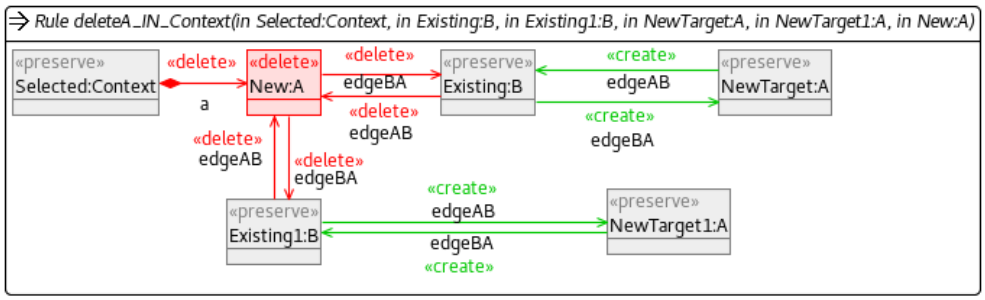
(c) Delete node A lower-bound repair single B aMPSO for m > n, m < *



(d) Delete node A lower-bound repair single B aMPSO for m > n, m = * and k = l = 1

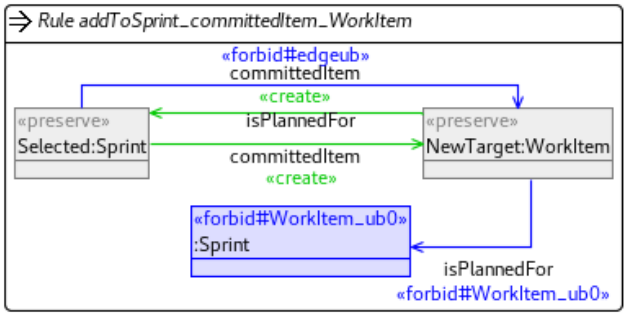


(e) Delete node A lower-bound repair many B forbid m A aMPSO for n = 2, m = 5 and k = 2, l = 2

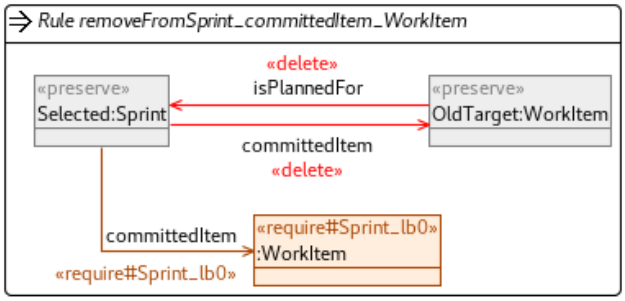


(f) Delete node A lower-bound repair many B aMPSO for n = 2, m = * and k = 2, l = 2

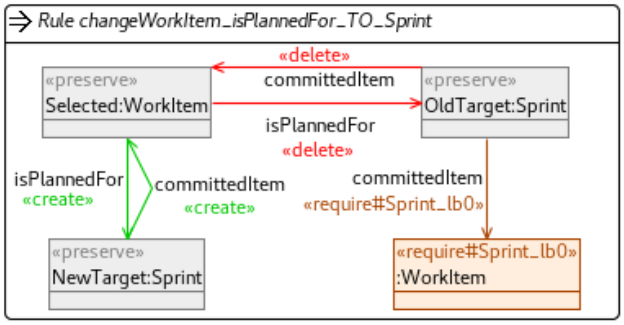
Fig. 10: Examples of generated delete node aMPSOs.



(a) Add Edge Rule



(b) Remove Edge Rule



(c) Change Edge Rule

Fig. 11: Generated edge manipulation aMPSOs for the Scrum Planning case study encoded as Henshin model transformations.

multiplicities for the edges between node B and nodes A and C. As a result of this refinement, there is a requirement that when a node B is created, it is connected to both node A and C, ensuring that their multiplicity constraints are satisfied.

Example iterative repair aMPSOs Fig. 16 shows two example generated iterative repair rules for node B. In Fig. 16a node B is created and the mandatory neighbours are lower-bound repaired, then in Fig. 16b node B is created and only node C is lower-bound repaired, while node A is connected to the created node B, with a NAC to ensure that no more than 2 nodes of type B are connected to it.

4.3.4 Generation algorithm completeness

In our approach, completeness refers to the ability of the aMPSOs to generate any of the supported modifications to the model. The changeable parts of a model are defined by the user, who can specify single or multiple nodes or edges which are allowed to be varied at search time. Our aMPSO generation algorithm produces the minimal set of atomic operations composed with the necessary repairs to ensure that the rules are applicable for any of the supported multiplicity patterns.

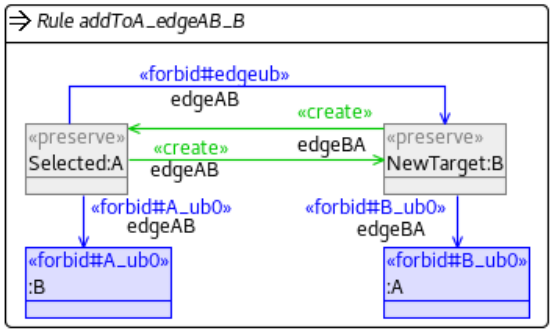
The generated aMPSOs form a complete set of rules that can reach any consistent variation of the changeable parts of a model through the application of a single operator or by chaining multiple operators. The generated aMPSOs can create or delete a node and add or remove edges between any two nodes that have the multiplicity pattern as shown in Fig. 6. These operations are enabled by complementing every node and edge creation or deletion with a corresponding repair to ensure that the operation is possible for any valid metamodel instance with respect to to the supported multiplicity patterns. The repairs also ensure that the resulting model instance does not invalidate the metamodel multiplicities after the application of the aMPSO.

4.3.5 Generation algorithm limitations

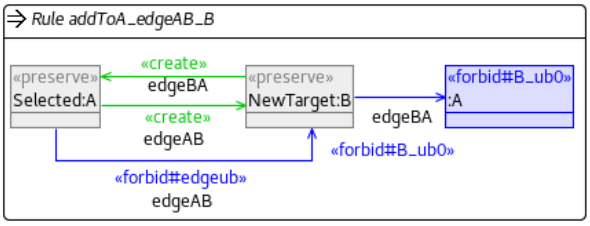
The rule generation algorithm presented in this chapter is aimed at producing atomic rules which can manipulate two nodes connected by an edge, as shown by the highlighted nodes A and B in the metamodel from Fig. 7. The described approach does not support the generation of more complex rules where additional context information is needed to generate a valid rule (*e.g.,* software refactoring patterns, feature model configuration). For such cases, the approach principles can be maintained (*e.g.,* create or delete node, add or remove edge), however, a domain-specific rule generation is required such that the additional domain information can be included in the generation algorithm.

4.4 Running search with aMPSOs

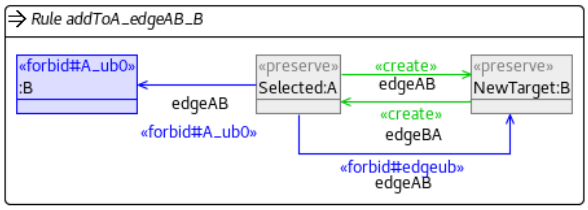
The algorithm proposed in the previous sub-section generates operators that preserve the consistency, with respect to to multiplicities, of the models modified. This addresses the first requirement on mutation operators that we identified in Sect. 4.1. It does not yet address the second requirement that mutation operators should work in both phases of an evolutionary search: phase 1, when some candidate solutions may not yet fully satisfy the solution-metamodel constraints, and phase 2 when all candidate solutions satisfy all solution-metamodel constraints. To satisfy this second requirement, we run the algorithm from Sect. 4.3 twice: First, we use it to generate rules based on problem-metamodel constraints. Next, we generate rules based on solution-metamodel constraints. We then use the union of the two sets of rules as the set of mutation operators for the evolutionary search.



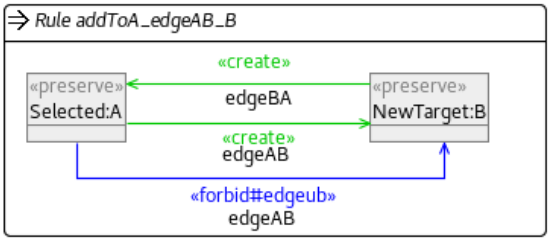
(a) Add edge NAC A B aMPSO for n = 0, m = 1 and k = 0, l = 1



(b) Add edge NAC B aMPSO for n = 0, m = * and k = 0, l = 1

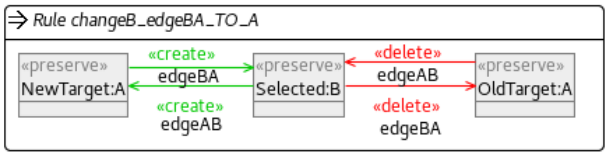


(c) Add edge NAC A B aMPSO for n = 0, m = 1 and k = 0, l = *

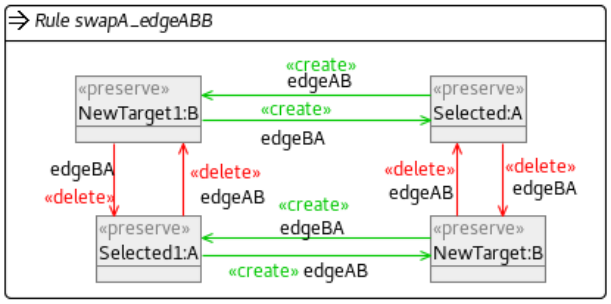


(d) Add edge aMPSO for n = 0, m = * and k = 0, l = *

Fig. 12: Examples of generated add edge aMPSOs.

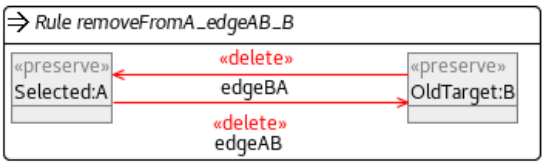


(a) Change edge aMPSO for n=0, m = 1 and k = l = 1

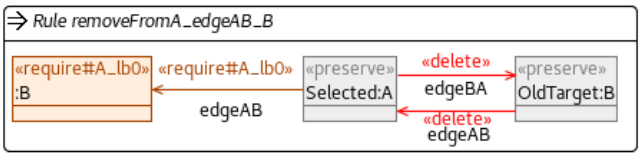


(b) Swap edge aMPSO for n = m = 1 and k= l = 1

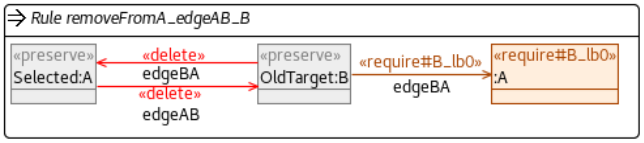
Fig. 13: Examples of generated change and swap edge aMPSOs.



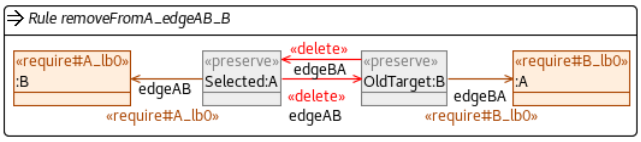
(a) Remove edge aMPSO for n = 0, m = 1 and k = 0, l = 1



(b) Remove edge PAC A aMPSO for n = 1, m = * and k = 0, l = 1



(c) Remove edge PAC B aMPSO for n = 0, m = 1 and k = 1, l = *



(d) Remove edge PAC AB aMPSO for n = 1, m = * and k = 1, l = *

Fig. 14: Examples of generated remove edge aMPSOs.

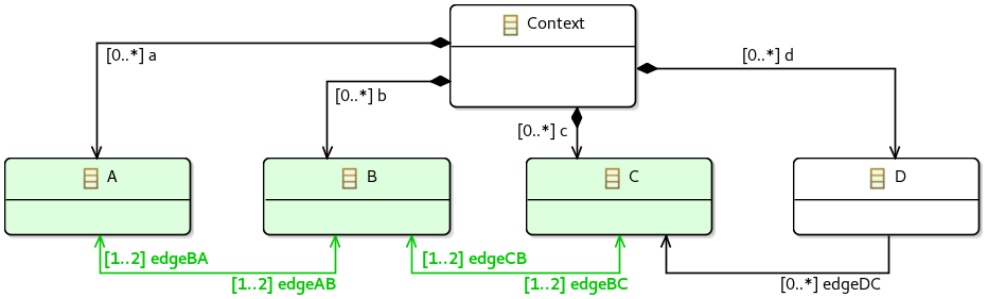
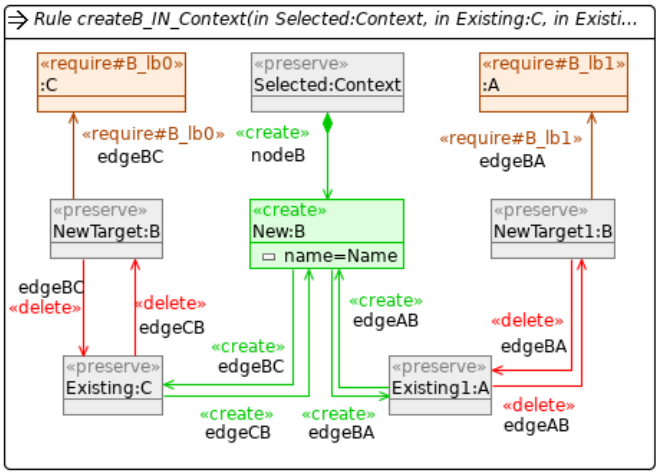
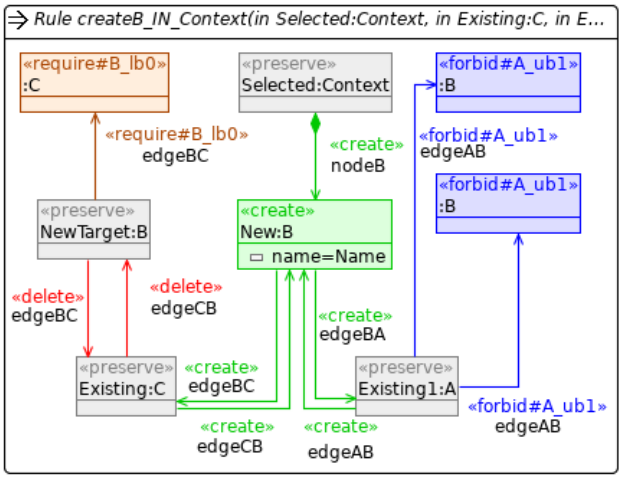


Fig. 15: Iterative repair metamodel. Nodes A, B and C have the multiplicity pattern shown in Fig. 6 for the connecting edges between them.



(a) Create node B, lower-bound repair C and lower-bound repair A



(b) Create node B, lower-bound repair C and connect to existing node A

Fig. 16: Examples of generated iterative node repair aMPSOs.

5 Experiments

To evaluate our rule generation approach, we seek to answer the following research question:

> *RQ*: What are the search quality and performance benefits of automatically generated mutation operators compared to mutation operators created manually?

To answer this research question, we ran experiments on 9 problem instances from 3 different case studies. These experiments aim to show that the generated mutation operators are at least as good as a set of operators created manually. The automatic generation of transformations is already an improvement over the manual process, as we remove the error-prone process of manual rule creation. Our evaluation aims to investigate whether there is a loss in search performance from generated operators. For each case study, we prepared a set of manually created mutation operators. Then, we configured MDEOptimiser to automatically generate mutation operators. Using both pairs of mutation operators, we ran experiments to solve the same problem instances. We compare the results from the two approaches to validate that the solutions obtained using the automatically generated aMPSOs are comparable with the results obtained using manually implemented search operators.

We do not include a comparison between our tool and other tools. Such a comparison isn't useful in this paper, as the focus is comparing the effectiveness and efficiency of the MAN and GEN operators. In [26] we compare the performance of MDEOptimiser and MOMoT, another model search tool that encodes search solutions as transformation chains.

5.1 Case studies

We selected a set of combinatorial optimisation problems that were implemented using MDEOptimiser. In the following subsections, we include a brief description of each case study.

5.1.1 Class-Responsibility Assignment

The Class Responsibility Assignment (CRA) case study was introduced at the 2016 Transformation Tool Contest (TTC)

Table 5: Summary of CRA input models.

Input Model	**A**	**B**	**C**	**D**	**E**
Attributes	5	10	20	40	80
Methods	4	8	15	40	80
Data Dependencies	8	15	50	150	300
Functional Dependencies	6	15	50	150	300

Table 6: Summary of input models for SP case study.

Input Model	**A**	**B**
Stakeholders	5	10
WorkItems	119	254
Backlog Size	455	1021

[19]. The goal of this problem is to transform a procedural software application to an object-oriented architecture while maintaining good cohesion and coupling. The quality of the produced solutions is measured using the CRA index defined in [19] as a single objective. The problem supplies a responsibility dependency graph that contains a set of functions and attributes with dependencies between them. In the metamodel, these entities are instances of the abstract type Feature.

To solve this problem, the user is required to create `Classes` in the `ClassModel` and assign `Features` to them such that: all `Features` are assigned to a `Class`; the model with the highest CRA index value is found. The problem has an additional constraint requiring that each Feature is assigned to only one `Class` at a time.

The CRA case study authors provide a set of five input models. The difference between them is the number of Features present. Model A is the smallest model with only nine features. The largest model provided is model E, with 160 features. Across all models, each set of features has an increasing number of dependencies between them. A summary of all the input models is included in Table 5.

We are specifying the CRA case study using two sets of transformations. The first set is implemented manually and consists of four operators as suggested in [10]. Other TTC'16 participants that used a similar approach to solve the case studies used similar rules [19, 37]. The second set of operators are aMPSOs generated using the approach presented in this paper.

5.1.2 Scrum Planning

We are running two experiments for the Scrum Planning (SP) case study described in Sect. 3. This case study has a similar problem specification as the CRA case study with the following differences: the assigned items do not have any dependencies between each other as `Features` do in the CRA case, and this case study is specified as a multi-objective problem.

In Table 6 we include a description of the input models used in experiments for this case study. These have been automatically generated by the authors using a random model generator. Through this case study evaluation, we explore how the difference in the number of objective functions affects the behaviour of manual and generated rules.

Table 7: Summary of input models for the NRP case study.

Input Model	**A**	**B**
Customers	5	25
Requirements	25	50
Software Artifacts	63	203

5.1.3 Next Release Problem

The goal of the Next Release Problem (NRP) is to find the optimal set of tasks to include in the next release for a software product, to minimise the cost and to maximise the customer satisfaction [45]. Each `Customer` has a desire which can consist of one or many `SoftwareArtifacts`. `SoftwareArtifacts` can have a recursive dependency on other `SoftwareArtifacts`.

To solve this problem, the user is required to assign instances of `SoftwareArtifacts` to a `Solution` such that the total cost of the selected `SoftwareArtifacts` is minimised and the total customer satisfaction is maximised.

We are specifying the next release problem using two sets of evolvers. One set was manually designed by the third author, who was not involved in developing the rule generation algorithm. The second set uses the automatically generated aMPSOs, using the approach described in this paper.

The minimal set of required rules for this case study is simple, only requiring mutations to add and remove an edge between a `Solution` and a `SoftwareArtifact`. However, the difference between this case study and the others considered in this paper is that the selection of a `SoftwareArtifact`, directly influences the Cost fitness function and indirectly influences the Customer Satisfaction objective. A `SoftwareArtifact` is considered for the calculation of a `RequirementRealization`, only when all its dependencies are also assigned to the solution. The set of evolvers manually designed for this case study, uses this additional information, ensuring that a *SoftwareArtifact* and all its dependencies are added in a single step. In contrast, the automatically generated aMPSOs don't use any additional problem information. With this difference, we aim to evaluate how the generated rules explore the search space in cases where the fitness functions provide only coarse–granular guidance.

The input models used for this case study have also been automatically generated by the authors using a random model generator. A brief description of these models has been included in Table 7.

5.2 Experiment configurations

We run experiments and compare the quality of the solutions obtained using two configurations: MAN with manually specified mutation operators and GEN with automatically generated mutation operators. For multi-objective problems, we use the hypervolume indicator and the ratio of best solutions for our comparison. For the CRA case, which is single–objective, we compare the quality of the solutions based on the median CRA score.

Experimental Setup All the experiments were repeated 30 times for statistical significance [5] and were executed on Amazon Web Services (AWS) c5.large spot instances running Amazon Linux 2 build 4.14.101-75-.76.amzn1.x86_64 running Java version 11.0.3+7-LTS. We ran our experiments using the NSGA-II algorithm [15], which is a well established evolutionary search algorithm that has been used successfully in many SBSE applications [9].

We undertook our experiments in two stages. The first stage was for determining ideal algorithm parameters (hyperparameters) that worked well for both configurations. The second stage used the hyperparameters found in the first stage and compared the quality of GEN with MAN.

Parameter Tuning We performed a systematic search for ideal population-size and number-of-evolutions hyperparameters that allow each configuration to find the best solutions. The combinations of analysed parameter configurations were each repeated 10 times to ensure robust results.

To identify a good number of evolutions to use in our experiments, we set the population size to 100 solutions. First, we analysed the growth of the median objective value for the single-objective problems and median hypervolume for the multi-objective problems. Then, we selected the number of evolutions after which there was no significant increase until the number of fitness evaluations has been exhausted, and the algorithm stopped. After we selected the number of evolutions based on the plateau of the fitness functions, we tried to reduce the size of the population by applying decrements of 25, until we reached a population size of 50. However, upon evaluating the results across all case studies, we determined the population size of 100 to be the best value for our experiments.

In Fig. 17 we show the evolution of the median CRA objective found by configurations with 5000 evolutions and population size 100. We observe that for all input models, except for E, the CRA index value plateaus after 2000 evolutions. For input model E, the GEN configuration continues to increase, even after the MAN configuration starts to plateau after passing 2000 evolutions.

Fig. 18 shows a summary of the hyperparameter runs for the SP case study. We observe that MAN is getting stuck in more than half of the experiment repetitions, leading to a median HV of 0 for this configuration. The GEN configuration is consistent at finding good solutions with a high HV value and starts to plateau after 2000 evolutions.

In Fig. 19 we show the evolution of the median HV by configurations with 5000 evolutions and population size 100 for the NRP case. We observe that there is no noticeable difference between MAN and GEN for input model A. However, for input model B, MAN finds a higher HV metric than GEN. We also observe that all configurations stop finding solutions after 500 evolutions for model A and 1000 evolutions for model B.

Based on the results of the experiment configurations discussed in this section, we selected the algorithm parameters for the experiment configurations used in our experiments. The selected number-of-evolutions parameter values are included in the Evol column in Table 9 for the CRA case, Table 11 for the SP case and Table 13 for the NRP case.

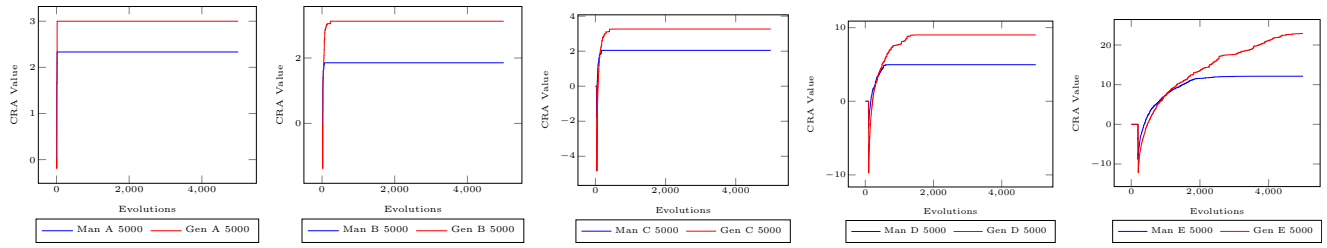
Hypervolume indicator Comparing solutions of optimisation problems that have more than one objective value is not a trivial problem. This is because when one objective value changes, the value of the other objectives can change as well. To overcome this problem, the hypervolume unary indicator has been proposed in [46]. This single-value metric measures the dominated volume between the solution points belonging to the Pareto front and a reference point (also nadir point) defined by the objective values of the worst solution. Higher hypervolumes indicate a Pareto front closer to the theoretical optimum.

Ratio of Best Solutions For multi-objective problems, we are calculating the *Best Solutions Ratio* (BSR) to show the number of non-dominated solutions contributed to the Pareto front by each configuration as presented by [22]. In our approach we are building the reference set (RS) using the best solutions found by all runs for both configurations. This metric allows us to measure the percentage of the contributions made to the reference set by each configuration.

$$BSR = \frac{|S \cap PF_{pseudo}|}{|PF_{pseudo}|} \tag{1}$$

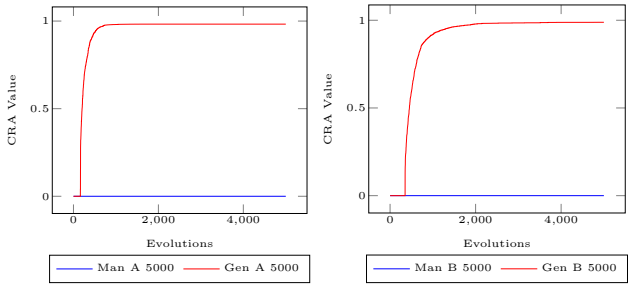
PF_{pseudo} stands for the reference set obtained by merging all the known nondominated solutions for a problem instance. S stands for the reference set of the configuration for which the metric is being calculated.

Statistical Analysis We use the Mann-Whitney U test to perform a statistical analysis of our results [33]. To measure the size of the differences between the configurations, we use Cohen's d effect metric [14]. We also include standard deviation (SD), skewness (Skew) and Kurtosis (Kurt) in our results tables to give a better indication of the solutions distribution found in our experiments.



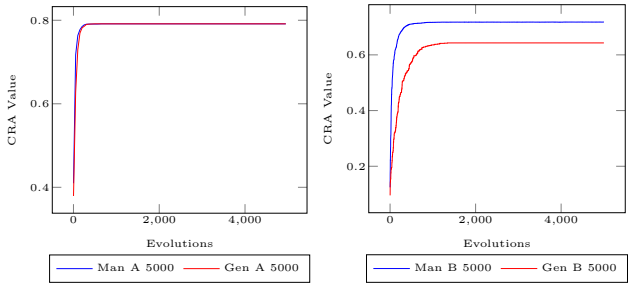
(a) Model A 5000 Steps (b) Model B 5000 Steps (c) Model C 5000 Steps (d) Model D 5000 Steps (e) Model E 5000 Steps

Fig. 17: Parameter search runs for the CRA case study. The X axis shows the number of algorithm steps, and the Y axis shows the median objective calculated across all batches ran for the parameter search.



(a) Model A 5000 Steps (b) Model B 5000 Steps

Fig. 18: Parameter search runs for the SP case study. The X axis shows the number of algorithm steps, and the Y axis shows the median hypervolume calculated across all batches ran for the parameter search.



(a) Model A 5000 Steps (b) Model B 5000 Steps

Fig. 19: Parameter search runs for the NRP case study. The X axis shows the number of algorithm steps, and the Y axis shows the median hypervolume calculated across all batches ran for the parameter search.

6 Results

In this section we present our experiment results for each of the case studies introduced in the previous section. We discuss each case study individually below. The complete data set can be downloaded from [12].

6.1 Class Responsibility Assignment

In Table 8 we list the mutation operators used for the two experiment configurations. Both configurations use three mutation operators to create a class (Create Class) and assign and change a feature (Assign Feature, Change Feature) with some small differences.

For GEN the Change Feature operator contains a PAC requiring that the source `Class` still has at least one `Feature` assigned following the application of this operator. At the same time, the MAN Change Feature operator can generate an empty class upon its application, and such instances are fixed by the delete empty class operator. In addition to these operators, GEN contains two additional operators to create and delete a `Class` after all the `Features` have been assigned. These ensure that the search does not get stuck in local optima in cases where the `Features` are assigned to too many or too few `Classes`.

Table 9 shows summary statistics for the CRA index found using the two configurations. In Fig. 20 we include

Table 8: Summary of CRA mutation operators for MAN and GEN.

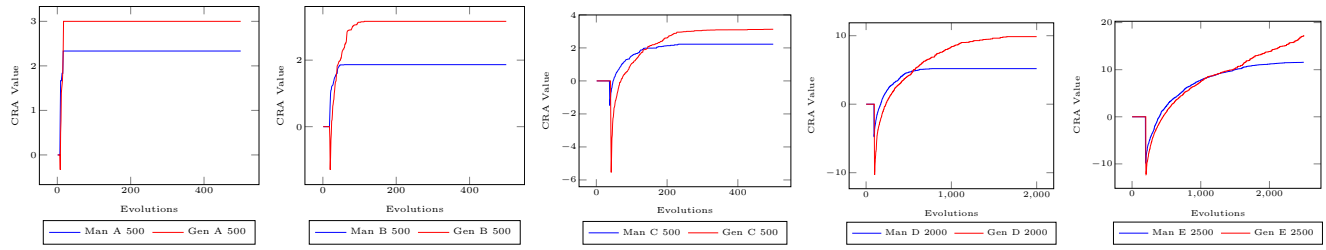
Manual	**Gen aMPSO**
Create Class	Create Class
N/A	Create Class Lb Repair
Assign Feature	Assign Feature
Change Feature	Change Feature
N/A	Remove Feature
Delete Empty Class	Delete Class
N/A	Delete Class Lb Repair

Table 9: CRA results for MAN and GEN. The Median column contains the median objective value across all experiment runs. For each model, the best solution is highlighted.

Config	**Evol**	**Median**	**Min**	**Max**	**SD**	**Skew**	**Kurt**
Man A	500	2.333	0.850	3.000	0.552	-0.679	-0.509
Gen A	500	**3.000**	3.000	3.000	0.000	0	0
Man B	500	1.865	1.238	3.104	0.514	0.642	-0.032
Gen B	500	**3.167**	1.826	4.083	0.599	-0.470	-0.376
Man C	500	2.224	1.148	3.240	0.572	-0.089	0.824
Gen C	500	**3.129**	2.110	3.806	0.428	-0.539	-0.039
Man D	2000	5.191	3.557	7.041	0.837	0.068	0.339
Gen D	2000	**9.863**	7.634	12.273	1.257	-0.176	0.782
Man E	2500	11.572	8.879	14.691	1.639	0.122	0.663
Gen E	2500	**17.323**	11.698	20.051	1.604	-1.106	-3.176

Table 10: Summary of statistical testing results for CRA.

	A	**B**	**C**	**D**	**E**
p-value	$<0.05\%$	$<0.05\%$	0.05%	0.05%	0.05%
U-value	795	809.5	817	900	884
Cohen's **d**	Large	Large	Large	Large	Large



(a) Model A 500 Steps (b) Model B 500 Steps (c) Model C 500 Steps (d) Model D 2000 Steps (e) Model E 2500 Steps

Fig. 20: Experiment results for the CRA case study. The X axis shows the number of algorithm steps, and the Y axis shows the median objective calculated across all experiment batches.

Table 11: SP results for MAN and GEN configurations.

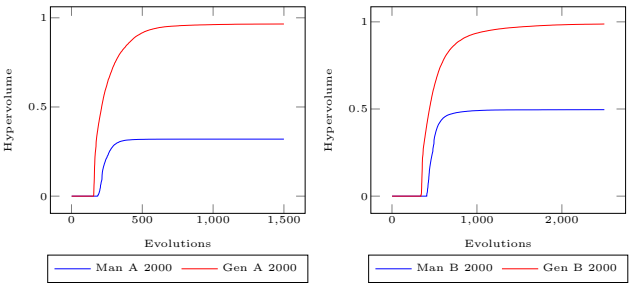
Config	Evol	Median	Min	Max	SD	RS	RSC	BSR
Man A	1500	0.000	0.000	0.960	0.460	13	0	0.00
Gen A	1500	**0.959**	0.957	0.995	0.010	13	13	**1.00**
Man B	2500	0.492	0.000	0.996	0.505	25	19	**0.76**
Gen B	2500	**0.988**	0.983	0.998	0.004	25	6	0.24

charts showing the median CRA value growth as the number of evolutions increases for both MAN and GEN configurations across all input models. For the smaller input models A, B and C, both MAN and GEN configurations find the highest objective value in the first 200 evolutions and plateau afterwards. For input model D, MAN does not find solutions that improve the CRA objective after 750 evolutions. The GEN configuration is slower than MAN at finding solution candidates with good objective values, however, after 500 evolutions, it finds a better CRA value than MAN and continues to find better solutions until reaching 2000 evolutions, after which it plateaus. For input model E, MAN finds a better CRA objective value slightly faster than GEN. After 1200 evolutions, GEN surpasses the MAN configuration and continues to find solutions with better CRA values, while MAN starts to plateau. For all input models, the configuration with automatically generated rules (GEN) consistently finds better median CRA index values than the configuration with manual rules (MAN). In all cases, GEN also finds higher minimum (Min) and maximum (Max) CRA scores than MAN. These results are confirmed by Table 10 which shows the p and U values of the Mann-Whitney test and Cohen's d effect size.

The quality of the results found by GEN is attributed to the aMPSO operators which allow classes to be created and deleted, after all the features have been assigned to a class, without invalidating the multiplicity constraints. The results for this experiment help us answer our *RQ* by showing that our approach is good at generating mutation operators, which lead to finding better solutions than the ones found using manually specified mutation operators.

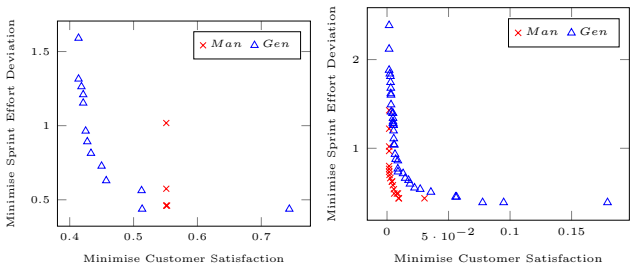
Table 12: Summary of SP mutation operators for MAN and GEN.

Manual	**Gen aMPSO**
Create Sprint	Create Sprint
N/A	Create Sprint Lb Repair
Add WorkItem	Add WorkItem
Change WorkItem	Change WorkItem
N/A	Remove WorkItem
Delete Empty Sprint	Delete Sprint
N/A	Delete Sprint Lb Repair



(a) Model A 1500 Steps (b) Model B 2500 Steps

Fig. 21: Experiment results for the SP case study. The X axis shows the number of algorithm steps, and the Y axis showsthe mean hypervolume calculated across all experiment batches.



(a) Input Model A (b) Input Model B

Fig. 22: Comparison of SP reference set Pareto fronts for MAN and GEN.

6.2 Scrum Planning

The SP case study is specified as a multi-objective problem. To compare the results, we will use the hypervolume metric. In Table 12 we include the mutation operators used for the two experiment configurations. Because the multiplicity pat-

tern between the `Sprint` and `WorkItem` metamodel entities is identical to the multiplicity between `Class` and `Feature`, the GEN mutation operators are similar to the ones generated for the CRA case study.

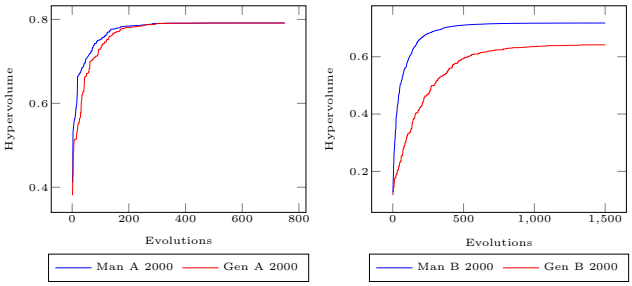
Table. 11 shows a comparison of the calculated hypervolumes for this case study for input models A and B. For both input models, MAN finds fewer constraint satisfying solutions, compared to GEN. For model A, MAN only found valid solutions in 10 out of the 30 experiment runs, compared to GEN, which found no invalid solutions. For cases where only invalid solutions have been found, we allocated a value of 0 for the hypervolume, because there are no constraint satisfying solutions generated, and the covered hypervolume space in those cases is 0. The same effect can be observed for model B, for which MAN finds valid solutions for 15 out of 30 experiments, compared to GEN, which always found a valid solution. Comparing the median hypervolume values for the configurations with valid solutions, we observe that GEN is better than MAN for both input models.

For model A, the highest hypervolume values have been found by GEN, and all the reference set contributions (RSC) are generated by this configuration, which has a BSR rate of 100%. In Fig. 22a we include the Pareto fronts found by the two configurations for model A, and in this figure, we can see that that GEN's reference set solutions dominates all the solutions found by MAN. The MAN reference set contains 5 solutions, while the GEN one has 13. The Mann-Whitney U test shows that GEN is better than MAN for model A (p=4.266E-6, U=761, Cohen's d=Large). For model B, GEN also found a higher median hypervolume value. However, for this model, MAN found 19 out of 25 reference set solutions, giving it a BSR rate of 76% while GEN only found 6 reference set solutions, with a BSR rate of 24%. The Mann-Whitney U test shows that GEN is as good as MAN for model B (p=0.25, U=527, Cohen's d=Large). In Fig. 22b we include the Pareto fronts found by GEN and MAN for model B. As indicated by the BSR rate, MAN finds more dominating solutions than GEN in the runs that found valid solutions, however, GEN found are more diverse solutions that cover a wider area along the Pareto curve. The MAN reference set contains 19 solutions, while the GEN one has 40 solutions.

We believe that MAN is getting stuck in local optima and the operators are unable to explore new solutions without temporarily invalidating or decreasing the quality of the current solutions. At the same time, GEN can explore new solutions without invalidating the constraints, but it is also affected by being stuck in local optima. We attribute the better results for model B found by MAN to the fact that after constraint satisfying solutions are discovered, the best solutions are found by moving `WorkItem` elements between `Sprints` until the right configuration is found.

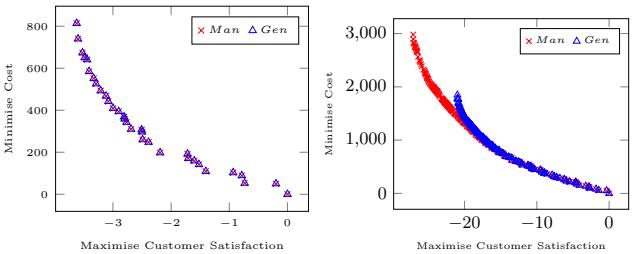
Table 13: NRP results for MAN and GEN.

Config	Evol	Median	Min	Max	SD	RS	RSC	BSR
Man A	750	0.791	0.791	0.791	0.000	32	32	1.00
Gen A	750	0.791	0.791	0.791	0.000	32	32	1.00
Man B	1500	**0.718**	0.712	0.722	0.003	281	281	**1.00**
Gen B	1500	0.641	0.635	0.643	0.002	281	63	0.22



(a) Model A 750 Steps (b) Model B 1500 Steps

Fig. 23: Experiment results for the NRP case study. The X axis shows the number of algorithm steps, and the Y axis shows the median hypervolume calculated across all experiment batches.



(a) Input Model A (b) Input Model B

Fig. 24: Comparison of NRP reference set Pareto fronts for MAN and GEN.

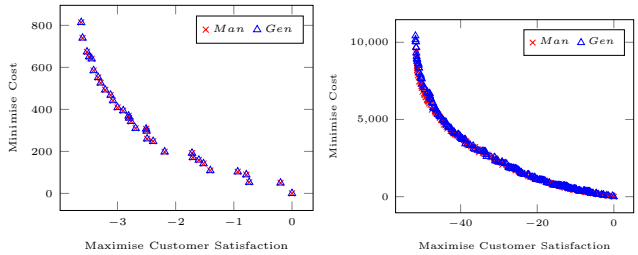
Table 14: Summary of NRP mutation operators for MAN and GEN.

Manual	Gen aMPSO
Modify Software Artifact	N/A
Modify SA With Dependencies	N/A
Assign Highest Realisation	N/A
Fix Dependencies	N/A
N/A	Add Software Artifact
N/A	Remove Software Artifact (PAC)

To answer our *RQ* for this case study, we observe that GEN finds a consistently good hypervolume, with a small SD value across all the repetitions, as seen in the Median and SD columns in Table. 11. The difference between the numbers of valid solutions found shows that the addition of the aMPSOs helped the search to find consistent solutions.

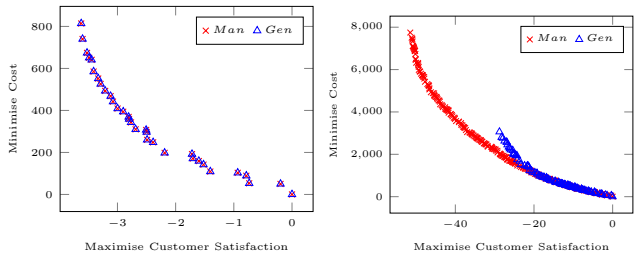
6.3 Next Release Problem

In contrast to the other use cases, the operators used in the NRP case are substantially different for both configurations (Table 14). The GEN operators only cover the basics: addition and removal of single `SoftwareArtifacts`. In both



(a) Input Model A (b) Input Model B

Fig. 25: Comparison of NRP reference set Pareto fronts for MAN and GEN with a mutation step size of 5 and recurrent mutation application.



(a) Input Model A (b) Input Model B

Fig. 26: Comparison of NRP reference set Pareto fronts for MAN and GEN with a mutation step size of 5 and non-recurrent mutation application.

situations, chances are that costs are raised without improving customer satisfaction due to the introduction of missing dependencies.

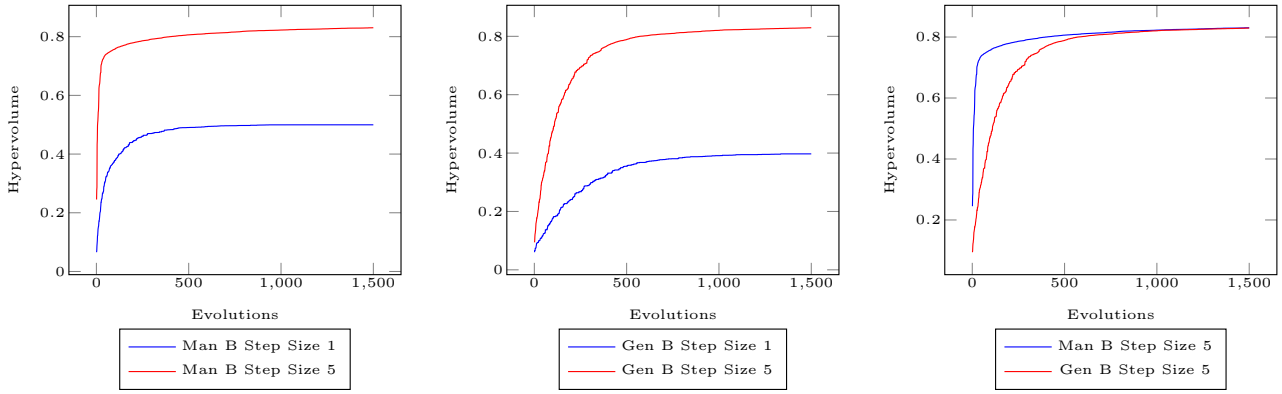
The first operator of MAN overcomes this problem, only adding a `SoftwareArtifact` if all of its dependencies are already part of the solution. Likewise, the removal of an artifact is only possible if no dependent artifacts are left over. The second operator allows for larger steps through the search space by adding and removing `SoftwareArtifacts` together with their direct dependencies and dependent artifacts, respectively. `Assign Highest Realisation` tries to exploit the fact that among `Realisations` of the same `Requirement` those with the highest `percentage` contribute most to the customer satisfaction. Considering all `Realisations` with yet unfulfilled dependencies, the operator selects the one with the highest `percentage` of fulfilment and adds its missing `SoftwareArtifacts` to the solution.

Note that none of the aforementioned operators take transitive dependency relations into account. Therefore, to counter the emergence of missing dependencies, the last operator is responsible for either adding all dependencies of an already selected `SoftwareArtifact` or removing the dependent artifacts of a formerly removed `SoftwareArtifact`.

For model A, we can observe that GEN consistently finds a hypervolume that is identical to MAN. Both configurations find the same solutions forming the reference set, and each has a BSR rate of 100%. We can also observe that the standard deviation metric between the hypervolumes across all 30 runs for each configuration is 0. This can also be observed in Fig. 24a which includes the identical Pareto fronts for both configurations. Because the solutions found are identical for this model, we are not including the statistical testing results in the paper. For model B, the hypervolume value found by MAN is higher than the one for GEN. However, on a closer inspection of the generated Pareto fronts for both configurations in Fig. 24b, we see that the solutions found by GEN, are subsets of the fronts for the MAN configuration. This is also confirmed by the data in Table 13. MAN has a BSR rate of 100%, while GEN only has a BSR rate of 22.4%. In this case, MAN also includes all the solutions generated by GEN. For model B the Mann-Whitney test shows at 5% confidence level that MAN finds solutions of better quality than GEN, with a large effect size.

We attribute this behaviour to how the MAN operators have been designed, compared to the ones automatically generated. The MAN operators are developed such that a `SoftwareArtifact`, together with all its dependencies, are all assigned to a Solution in a single application. At the same time, the GEN operators assign `SoftwareArtifacts`, one by one, by adding or removing edges, in atomic operations. Because the Customer Satisfaction objective does not guide the solutions, unless all `SoftwareArtifacts` realising a complete Realisation are assigned, GEN is slower at finding converging solutions, requiring more evolutions. However, comparing the structure of the operators, a single operation of the manual operator, which moves a `SoftwareArtifact` together with all its dependencies in a single evolution, is equivalent to multiple applications of the Gen operator for adding an edge.

This suggests that we may be able to overcome GEN's deficiency by allowing multiple GEN operators to be applied in each algorithm step. We refer to the number of applications as the 'step size'. Allowing the search to apply multiple mutation steps to derive an offspring model allows the generated mutation operators to outperform the manually constructed ones. This is remarkable given that the generated operators are much simpler than the manually constructed ones for the NRP case study. However, it is worth noting that the success of this approach depends on how the operators are applied: recurrent application of the same operator multiple times is helpful, non-recurrent application, consisting of randomly choosing a new operator for each application is not. This behavior can be observed in the reference set solutions found by each mutation application strategy, as can be seen in Fig. 25 and Fig. 26. For this case study the recurring mutation application strategy is better than the non-recurring one.



(a) NRP Input Model B MAN results comprison using step size 1 and 5

(b) NRP Input Model B GEN results comprison using step size 1 and 5

(c) NRP Input Model B results comprison between MAN and GEN using step size 5

Fig. 27: Pairwise comparison of NRP hypervolume between MAN and GEN configurations with a mutation step size of 1 and a mutation step size of 5 using recurrent mutation application. The X-axis shows the number of algorithm steps, and the Y-axis shows the median hypervolume calculated across all experiment batches.

In Fig. 27 [3] we include charts that compare the median hypervolume between MAN and GEN with a step size of 1 and MAN and GEN with a step size of 5, using recurring mutation application. For input model A, the additional step size does not lead to finding better solutions for either configuration. Therefore we do not include the charts for this model.

For input model B, MAN can find a better median hypervolume with the increased step size. The GEN configuration also finds better solutions with the increased step size and can find identical solutions to the ones found by MAN. Comparing the HV of GEN with step size 5 and MAN with step size 1, we can see that by using the aMPSOs combined with increased step size, the search can find results that are better than the configuration using more complex manually designed mutation operators.

Fig. 27c shows that MAN finds a better hypervolume faster than GEN when either set of operators are used with a step size of 5. This effect appears because the MAN operators perform more than one atomic operation in a single application, and by increasing the step size, the number of performed operations also increases, leading to faster convergence. The GEN configuration needs more time to identify the ideal order of mutations to apply to search solution candidates to find the same solutions. Both configurations plateau after 1000 evolutions. An in-depth analysis of this effect is out-of-scope for the present paper. We are working on a detailed analysis of different mutator selection and application strategies, which will be presented in a future publication.

These results help us answer our *RQ* for this case study by showing that atomic operators generated using our approach can be just as good as more complex manually created mutations.

6.4 Search Operators Efficiency Comparison

We also compared the efficiency of the generated operators with the manually created ones. In the NRP case study, the generated operators led to shorter (or at most equal) average runtimes for the ES. For the CRA and SP scenarios, the generated rules were less efficient than the manual ones.

In Table 15 we include a runtime summary for the two configurations we are evaluating across all input models for the CRA case study. We observe a higher runtime for GEN configurations, that is almost double the time required by the MAN configuration. We attribute this difference to the higher number of rules used by the GEN configuration. We differentiate two different matching strategies in MDEOptimiser: the *classic strategy*[4] first finds all possible matches for all operators and then uniformly randomly selects one of them, while the *non-deterministic matching strategy* uses Henshin's non-deterministic matching algorithm by uniformly randomly selecting one mutation operator and then letting Henshin apply this for a random match. For the CRA case, there are more generated operators than manual ones, which means that more matches must be generated in the 'classic strategy'. As a result, under this strategy, the search with generated rules took up to approximately 3 times as long as with manual rules as shown in Table 15. With the 'non-de-

[3] The charts in this figure show hypervolume for the step size 1 metric calculated against a reference point that includes the additional solutions found by the increased step size strategies. This causes the step size 1 hypervolume metric to be lower than those reported in Table 13 because the step size 1 reference set covers a smaller portion of the step size 5 reference set.

[4] Which the tool authors used in their submission to TTC'16 [10]

Table 15: Summary of CRA elapsed time in seconds for the MAN and GEN configurations across all input models using 'classic' matching.

Time	Man A	Gen A	Man B	Gen B	Man C	Gen C	Man D	Gen D	Man E	Gen E
Mean	15.10	27.90	23.27	43.76	41.32	75.28	611.40	1177.70	2972.65	4298.16
Median	14.92	27.61	22.04	44.24	41.07	75.65	590.75	1188.91	2869.16	4198.20
Min	11.44	26.48	17.33	33.92	26.79	64.19	452.90	991.29	2193.35	3582.67
Max	17.64	30.09	34.99	50.13	58.43	86.35	853.47	1416.96	4202.11	5189.39

Table 16: Summary of SP elapsed time in seconds for the MAN and GEN configurations across all input models using 'non-deterministic' matching.

Time	Man A	Gen A	Man B	Gen B
Mean	119.13	291.25	484.90	3069.18
Median	120.67	291.87	487.76	3016.74
Min	107.63	261.25	447.16	2686.92
Max	130.47	367.78	510.77	4171.07

Table 17: Summary of NRP elapsed time in seconds for the MAN and GEN configurations across all input models using 'non-deterministic' matching.

Time	Man A	Gen A	Man B	Gen B
Mean	275.42	223.42	1677.80	1355.29
Median	274.96	224.27	1676.22	1348.97
Min	258.84	215.79	1610.85	1312.45
Max	307.71	234.52	1813.63	1412.93

terministic matching strategy', the generated rules led to a faster search than the manual rules.

In Table 16 we include a summary of the runtime for the SP case study. We observe that GEN is slower than MAN. After closely inspecting the generated results, we observed that GEN finds more constraint satisfying solutions, and more time is spent evaluating the fitness functions. At the same time, the NSGA-II archive contains more solutions for the GEN configuration compared to MAN, which results in more time being required to perform the required domination and crowding comparisons. This leads to an increase in the runtime for GEN.

Table 17 shows the runtime summary for the NRP case study. For both input models in this case study, the GEN configuration reaches the termination condition in less time than MAN. This time difference is caused by the MAN operators, which are more complex than the GEN ones (Add Software Artifact and Remove Software Artifact) and require more time to be applied.

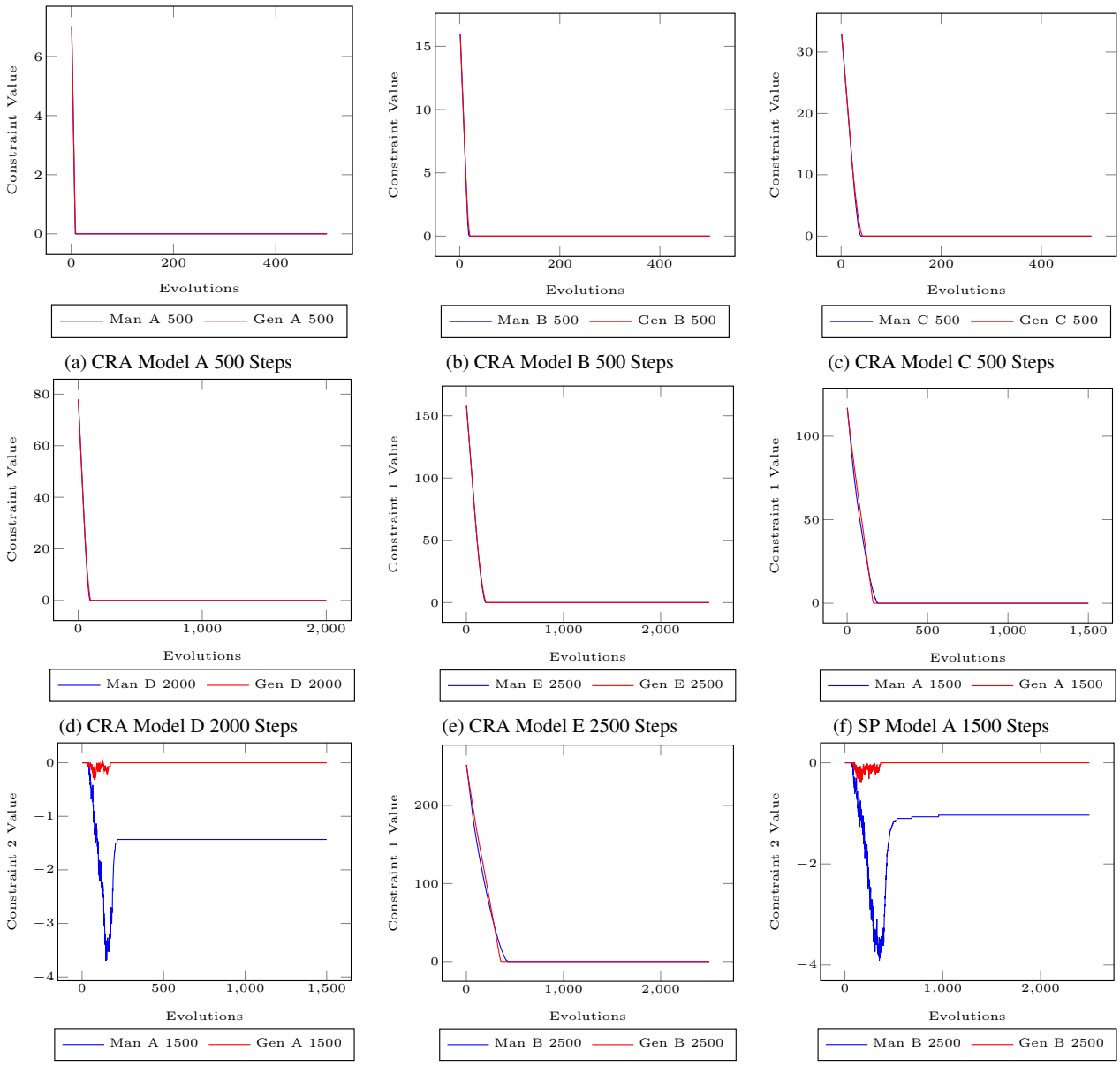
In Figure 28, we show the values returned by each individual problem constraint function for the MAN and GEN configurations in the CRA and SP case studies. The values shown are the medians for each problem constraint function (one constraint function for CRA and two constraints for SP) over the 30 experiment repetitions. A solution candidate is considered feasible with respect to the problem constraint functions when the constraint functions return a value of 0. A value returned by a constraint function, different from 0, indicates how far a solution is from satisfying the constraint. The metrics are calculated using the solutions in the algorithm archive. Across all the CRA problem instances, we can observe that the number of steps required to satisfy the problem constraint (minimise the number of features not assigned to a class) is identical for all problem instances for both MAN and GEN. For input model E, GEN is slightly faster at finding solutions that satisfy the problem constraint, however, the difference in the number of steps is small. We attribute this effect to the higher number of mutation operators in the GEN configuration, which can improve the problem constraint.

In the SP case study, for Constraint 1, both MAN and GEN require a similar number of steps to find satisfying solutions. As in the case of the CRA problem instances, GEN is slightly faster at finding solutions that satisfy Constraint 1 for both problem instances in this case study. For Constraint 2, as shown in Figs 28g for input model A and Fig. 28i for input model B, MAN is unable to find constraint satisfying solutions and for some experiments it gets stuck in local optima. This is indicated by the flat median constraint value observed in the charts.

6.5 Threats to validity

The validity of the conclusions we draw from our data depends on a number of factors: 1. the degree to which the selected case studies are representative of real-world problems, 2. the chosen hyperparameters (e.g., population size, number of evolutions, ...), 3. the degree to which the chosen input models are realistic and representative of real-world problems, and 4. the provenance of the manual rules used in our experiments.

We used a varied selection of case studies that cover both single and multi-objective scenarios and allow a systematic exploration of different aspects of the overall problem. All hyperparameter values were selected systematically to ensure that no approach is favoured over the other. We applied the recommended steps to ensure that our results are accurate and correctly interpreted and described [5]. Input models were either provided as part of pre-existing case studies (CRA [19]), or were randomly generated, ensuring consistency with the given problem metamodel. Recently, better model generators have been proposed [40, 41] that aim to produce more realistic model instances for such evaluations. We are interested in exploring the use of such generators for



(a) CRA Model A 500 Steps (b) CRA Model B 500 Steps (c) CRA Model C 500 Steps

(d) CRA Model D 2000 Steps (e) CRA Model E 2500 Steps (f) SP Model A 1500 Steps

(g) SP Model A 1500 Steps (h) SP Model B 2500 Steps (i) SP Model B 2500 Steps

Fig. 28: Comparison of the median constraint values for the CRA and SP case studies. The X axis shows the number of algorithm steps, and the Y axis shows the median constraint value calculated across all experiment batches for the specific problem instance.

further evaluation of our approach. Finally, the manual rules that we used in our experiments were all produced without consideration of the generative principles we propose in this paper: the CRA rules were produced by the authors in 2016 [10], well before we started considering the automatic generation of rules; the SP rules are very similar to the CRA rules. The 3rd author, who was not involved in the design of the rule generation algorithm, produced the NRP rules taking into account the structure of the objective functions during rule construction.

7 Related work

7.1 Model-Based Optimisation

Moawad *et al.* [35] proposed Polymer, an approach that runs search directly over models using an optimisation framework that is built as an extension of the Kevoree modelling framework [20]. Polymer offers interfaces for implementing variation operators and fitness functions. Compared to our approach Polymer does not use model transformations to evolve models. Instead, variation operators and fitness func-

tions are executed directly over models and are implemented as Java classes by the user, making use of domain-specific entities specified in the metamodel. Specifying mutation operators using Java, however, makes it more difficult to automatically generate them.

Fleck *et al.* [18] proposed Marrying Optimisation and Model Transformations (MOMoT), a tool that optimises sequences of model transformation rules applications. MOMoT is implemented as an Eclipse plugin and allows users to specify optimisation problems using a flexible DSL. The model transformations used by MOMoT are encoded using Henshin. The tool uses mutation and crossover search operators applied to the search solution candidates encoded as sequences of transformation rules applications. An additional repair step is required during the search to ensure that after the application of search operators, the resulting rules applications sequences produce feasible solutions. The optimisation algorithms used by MOMoT are implemented using MOEAFramework. Abdeen *et al.* [2] proposed a multi-objective, rule-based design space exploration (DSE) framework built using the ViatraDSE framework [24]. The multi-objective implementation uses the NSGA-II algorithm to find efficient model derivation chains. The framework uses mutation and crossover operators, and a repair mechanism is implemented to ensure that the resulting sequences of transformation rules applications produce feasible solutions. Infeasible rule applications sequences are truncated or discarded if they cannot be applied to the initial model. Because they encode search solution candidates as sequences of transformation rules applications, both ViatraDSE and MOMoT can use mutation and crossover search operators, while in MDEOptimiser, which searches directly over models, only mutation is currently supported. ViatraDSE and MOMoT have the advantage that they can be more memory efficient compared to model-based approaches because they only keep a sequence of transformations to be applied to an initial model compared to individual copies of models. One drawback of this approach is that for each fitness evaluation, the sequence of transformations (sometimes along with potential repair operations) has to be applied to obtain a phenotype for evaluation. Compared to MDEOptimiser, ViatraDSE and MOMoT do not offer any mutation operator generation, and users are required to provide model transformations to be used for search space exploration. John *et al.* [26] evaluate the performance of model-based and rule-based search by comparing MDEOptimiser and MOMoT.

Other uses of model-based search techniques are the approaches used to solve the problem of metamodel/model co-evolution, either automatically [30] or interatively [29, 31]. In their automated approach, Kessentini *et al.* [30] require the user to provide as inputs the intial and revised metamodels, a set of input models conforming to the original model and a set of allowed edit operations. The generated outputs consist of minimal sequences of transformations to apply to the input models so they conform to the revised metamodels. In the interactive metamodel/model evolution approach [29], the authors use clustering to reduce the number of feasible search solutions and use human input to guide the search towards preferred solutions.

Mutation Generation The generation of mutation operators for evolutionary algorithms has been studied in the wider optimisation literature. To the best of our knowledge, FitnessStudio [42] is the only approach in an MDE context. FitnessStudio is a meta-learning tool for generating in-place model transformation rules that can be used as search operators in model-based optimisation. The algorithm generates mutation operators that obtain good results for the CRA case study [19]. The main drawback is that the user is required to first execute a learning operation on a test model and then run the optimisation with the generated rules on the rest of the models that have to be optimised. The effectiveness of the approach depends on the model used for learning, its coverage of the metamodel and its similarity to the remaining models. In contrast to FitnessStudio, our approach does not require the additional meta-learning step.

Hong *et al.* [25] present an offline hyper-heuristic approach that automatically generates mutation operators using genetic programming and meta-learning. These are then used in evolutionary programming to solve several test functions. This technique requires an already existing genetic encoding of the problem. In contrast, we support problems that are naturally encoded in a suitable domain-specific modelling language. The work of Hong *et al.* [25] is similar to the work of Strüber [42], requiring a training step to first generate the mutation operators, which are then used to solve other problems. Our algorithm generates the mutation operators using the problem specification and does not require a training step.

Alhwikem *et al.* [4] introduce an approach for generating mutation operators for MDE languages. The goal of this approach is to use the generated operators to generate test inputs when performing mutation analysis. The systematically generated mutations can be used to change features of Ecore based models by adding, removing or changing values of a model feature in order to increase test coverage. The atomic mutations generated by this approach are similar to some operations we generate for the simple cases, namely to add, remove and change an element. In addition to these operators, our approach also generates more complex repairs.

Mengerink *et al.* [34] propose a methodology for creating a complete DSL operator library for evolving EMF-based languages. The operators are atomic, and the proposed list of the most used operators, based on their occurrence in the DSL evolution history, includes the aMPSO and MPSO operators generated by our approach and the SERGe rules

generator. This library aims to be a complete list of all the possible atomic mutation operators for EMF-based languages. The important contribution we make is to selectively generate only those operators useful in the context of ES.

Kosiol *et al.* [32] propose a classification and formalisation for graph transformations based on the effect they have on the graph constraints: consistency sustaining transformations manipulate graphs without decreasing the number of satisfied constraints; consistency improving transformations, seek to reduce the number of graph constraint violations. The formalisation proposed by the authors enables precise reasoning about the behaviour of graph transformations used as search operators. Kosiol *et al.* [32] also provide static analysis techniques for checking if a transformation rule is sustaining or improving. It would be interesting to apply this analysis to the rules generated by our approach.

Mutation Weighting Doerr *et al.* [16] propose the use of operator strengths to increase the degree of changes performed by an operator. The authors show that a combination of atomic changes combined with variably sized changes, is the best for increasing the speed of solving optimisation problems. Using only atomic operators, the search can be slow, requiring many steps to be performed, while using operators that perform bigger changes, the search can have difficulty in finding neighboring solutions that have better fitness. The case study evaluation presented in this paper showed that this is also a problem affecting our approach. For NRP and the case study presented by Murphy *et al.* [36], the generated aMPSOs require more applications to find good solutions, compared to operators that perform multiple operations in a single step. One potential solution to this problem is increasing the number of evolutions, and at the expense of longer runtime, the search can find better solutions if the fitness functions can efficiently guide each aMPSO application. Alternatively, the problem can be solved using a combination of operators consisting of aMPSOs and compositions of multiple aMPSOs that are applied in a single step. Our initial experiments using this approach for the NRP case study, in which we applied 5 aMPSOs in a single evolution step, have shown promising results. This is a problem we seek to explore in future work to improve the performance of aMPSOs generated using our approach.

8 Conclusions

We showed how mutation operators for search-based model engineering can be generated automatically without the need for meta-learning. The efficiency and effectiveness of the atomic multiplicity-preserving search operators (aMPSOs) we generate are comparable to search operators manually specified by expert users (and better in some cases). However, automatic generation requires less human effort and reduces the risk of accidentally introduced errors.

Our generated rules coped well with single- and multiple-objective problems as well as with a problem where the objective function provides only fairly coarse-grained guidance to the search. However, improvements are clearly possible. In particular, in our future work, we plan to investigate the following questions:

- We validated the correctness of our approach by using a suite of unit tests to check that the generation algorithm produces the expected output for each supported scenario. Formalising our approach, along with providing a correctness and completeness proof, are left for future work.
- In the CRA case study, we saw that the startup behaviour of our generated rules differs from that of the manual rules, such that the manual rules find better solutions in early evolutions. We will study what affects this startup behaviour and how we may be able to improve our generated rules in this area. For example, it may be useful to use separate sets of rules for the two phases of the search (cf. Sect. 4.1) to ensure more focused exploration during the first phase.
- Optimisation problems use other constraints beyond multiplicities. Arbitrary constraints are difficult to handle without additional user input; however, specific types of constraints or constraint templates can be more easily incorporated. For example, we are currently working on implementing rule generation for feature-model constraints.
- Recursive repair offers additional opportunities for repair in MPSOs, but at the cost of higher generation effort and a larger set of search operators. Which, if any, recursive repair strategies offer benefits to the overall search?
- Some problems, including some of the examples discussed in this paper, may be solvable by optimising constraint solvers. However, constraint solvers work off a relatively low-level problem encoding, making them more difficult to use. We are interested in understanding if, and under what circumstances, it is possible to translate automatically from our high-level problem representation into an encoding that can be efficiently solved by a constraint solver.

Acknowledgements This work has been supported by the *Engineering and Physical Sciences Research Council* (EPSRC) under grant number 1805606.

References

1. Choco-solver. `https://choco-solver.org/`. Accessed: 2020-11-04

2. Abdeen, H., Varró, D., Sahraoui, H., Nagy, A.S., Debreceni, C., Hegedüs, Á., Horváth, Á.: Multi-objective optimization in rule-based design space exploration. In: Proceedings of International Conference on Automated Software Engineering, pp. 289–300. ACM (2014). DOI 10.1145/2642937.2643005
3. Aleti, A., Moser, I.: A Systematic Literature Review of Adaptive Parameter Control Methods for Evolutionary Algorithms. ACM Computing Surveys **49**(56) (2016). DOI 10.1145/2996355
4. Alhwikem, F.H.M., Paige, R.F., Rose, L.M., Alexander, R.D.: A systematic approach for designing mutation operators for MDE languages. In: Proceedings of Workshop on Model-Driven Engineering, Verification and Validation, vol. 1713, pp. 54–59. CEUR (2016)
5. Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability **24**(3), 219–250 (2014). DOI 10.1002/stvr.1486
6. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Proceedings of International Conference on Model Driven Engineering Languages and Systems, pp. 121–135 (2010). DOI 10.1007/978-3-642-16145-2_9
7. Bézivin, J.: Model driven engineering: An emerging technical space, pp. 36–64 (2005). DOI 10.1007/11877028_2
8. Biermann, E., Ermel, C., Taentzer, G.: Formal foundation of consistent emfmodel transformations by algebraic graphtransformation. Software & Systems Modeling **11**(2), 227–250 (2012)
9. Boussaïd, I., Siarry, P., Ahmed-Nacer, M.: A survey on search-based model-driven engineering. Automated Software Engineering **24**(2), 233–294 (2017). DOI 10.1007/s10515-017-0215-4
10. Burdusel, A., Zschaler, S.: Model optimisation for feature class allocation using MDEOptimiser: A TTC 2016 submission. In: Proceedings of 9th Transformation Tool Contest, pp. 33–38. CEUR (2016)
11. Burdusel, A., Zschaler, S.: Towards Scalable Search-Based Model Engineering with MDEOptimiser Scale. In: Proceedings of Workshop on Artificial Intelligence and Model-Driven Engineering, pp. 189–195. IEEE (2019). DOI 10.1109/models-c.2019.00032
12. Burdusel, A., Zschaler, S., John, S.: Automatic generation of atomic consistency preserving searchoperators for search-based model engineering - accompanying data. `http://dx.doi.org/10.6084/m9.figshare.12284468`. DOI 10.6084/m9.figshare.12284468
13. Burdusel, A., Zschaler, S., John, S.: Automatic generation of atomic consistency preserving search operators for Search-Based model engineering. In: Proceedings of International Conference on Model Driven Engineering Languages and Systems, pp. 106–116. IEEE (2019). DOI 10.1109/models.2019.00-10
14. Cohen, J.: Statistical power analysis for the behaviors science, 2 edn. Laurence Erlbaum Associates (1988)
15. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation **6**(2), 182–197 (2002). DOI 10.1109/4235.996017
16. Doerr, B., Doerr, C., Kötzing, T.: The right mutation strength for multi-valued decision variables. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1115–1122. ACM (2016)
17. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing, 2nd edn. Springer Publishing Company, Incorporated (2015). DOI 10.1007/978-3-662-44874-8
18. Fleck, M., Troya, J., Wimmer, M.: Search-based model transformations. Journal of Software: Evolution and Process **28**(12), 1081–1117 (2016). DOI 10.1002/smr.1804
19. Fleck, M., Troya, J., Wimmer, M.: The Class Responsibility Assignment Case. In: Proceedings of 9th Transformation Tool Contest, vol. 1758, pp. 1–8. CEUR (2016)
20. Fouquet, F., Nain, G., Morin, B., Daubert, E., Barais, O., Plouzeau, N., Jézéquel, J.M.: An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements. In: Proceedings of International Conference on Model Driven Engineering Languages and Systems, pp. 87–101. Springer (2012)
21. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: E ssence: A constraint language for specifying combinatorial problems. Constraints **13**(3), 268–306 (2008)
22. Hansen, M.P., Jaszkiewicz, A.: Evaluating the quality of approximations to the non-dominated set. Tech. Rep. IMM-REP-1998-7, Technical University of Denmark, Department of Mathematical Modelling (1998)
23. Harman, M., Jones, B.F.: Search-based software engineering. Information and Software Technology **43**(14), 833–839 (2001). DOI 10.1016/S0950-5849(01)00189-6
24. Hegedüs, Á., Horváth, Á., Ráth, I., Varró, D.: A model-driven framework for guided design space exploration. In: Proc 26th IEEE/ACM Int'l Conf. Automated Software Engineering (ASE'11), pp. 173–182 (2011). DOI 10.1109/ASE.2011.6100051
25. Hong, L., Drake, J.H., Woodward, J.R., Özcan, E.: A hyper-heuristic approach to automated generation of mutation operators for evolutionary programming. Applied Soft Computing **62**, 162 – 175 (2018). DOI 10.1016/j.asoc.2017.10.002
26. John, S., Burdusel, A., Bill, R., Struber, D., Taentzer, G., Zschaler, S., Wimmer, M.: Searching for optimal models: Comparing two encoding approaches. In: Proceedings of International Conference on Model Transformations (2019). DOI 10.5381/jot.2019.18.3.a6
27. Kehrer, T.: Calculation and propagation of model changes based on user-level edit operations: a foundation for version and variant management in model-driven engineering. Ph.D. thesis, University of Siegen (2015). URL `http://dokumentix.ub.uni-siegen.de/opus/volltexte/2015/963/`
28. Kehrer, T., Taentzer, G., Rindt, M., Kelter, U.: Automatically deriving the specification of model editing operations from metamodels. In: Proceedings of International Conference on Theory and Practice of Model Transformations, pp. 173–188 (2016). DOI 10.1007/978-3-319-42064-6_12
29. Kessentini, W., Alizadeh, V.: Interactive metamodel/model co-evolution using unsupervised learning and multi-objective search. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems, pp. 68–78 (2020)
30. Kessentini, W., Sahraoui, H., Wimmer, M.: Automated metamodel/model co-evolution: A search-based approach. Information and Software Technology **106**, 49–67 (2019)
31. Kessentini, W., Wimmer, M., Sahraoui, H.: Integrating the designer in-the-loop for metamodel/model co-evolution via interactive computational search. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems, pp. 101–111 (2018)
32. Kosiol, J., Strüber, D., Taentzer, G., Zschaler, S.: Graph consistency as a graduated property: Consistency-Sustaining and -Improving graph transformations. In: Proceedings of International Conference on Graph Transformation, pp. 189–195. Springer (2020). DOI 10.1007/978-3-030-51372-6_14
33. Mann, H.B., Whitney, D.R.: On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Statist. **18**(1), 50–60 (1947). DOI 10.1214/aoms/1177730491
34. Mengerink, J.G.M., Serebrenik, A., Schiffelers, R.R.H., v. d. Brand, M.G.J.: A Complete Operator Library for DSL Evolution Specification. In: Proceedings of the International Conference on Software Maintenance and Evolution, pp. 144–154. IEEE (2016). DOI 10.1109/ICSME.2016.32
35. Moawad, A., Hartmann, T., Fouquet, F., Nain, G., Klein, J., Bourcier, J.: Polymer: A Model-Driven Approach for Simpler, Safer, and Evolutive Multi-Objective Optimization Development. In: Proceedings of International Conference on Model-Driven Engineering and Software Development, pp. 1–8. IEEE (2015)

36. Murphy, J., Burdusel, A., Michael, L., Zschaler, S., Black, E.: Deriving persuasion strategies using search-based model engineering. In: Proceedings of International Conference on Computational Models of Argument, vol. 305, pp. 221–232. IOS Press (2018)
37. Nagy, A.S., Szárnyas, G.: Class Responsibility Assignment Case: a Viatra-DSE Solution. In: Proceedings of the 9th Transformation Tool Contest, pp. 39–344. CEUR (2016)
38. Nassar, N., Radke, H., Arendt, T.: Rule-Based Repair of EMF Models: An Automated Interactive Approach. In: Proceedings of International Conference on Theory and Practice of Model Transformations, pp. 171–181. Springer (2017). DOI 10.1007/978-3-319-61473-1_12
39. Rubin, K.S.: Essential Scrum. Addison-Wesley (2012)
40. Schneider, S., Lambers, L., Orejas, F.: Symbolic model generation for graph properties. In: Proceedings of International Conference on Fundamental Approaches to Software Engineering, pp. 226–243. Springer (2017). DOI 10.1007/978-3-662-54494-5_13
41. Semeráth, O., Nagy, A.S., Varró, D.: A graph solver for the automated generation of consistent domain-specific models. In: Proceedings of the International Conference on Software Engineering, pp. 969–980. ACM (2018)
42. Strüber, D.: Generating efficient mutation operators for Search-Based Model-Driven engineering. In: Proceedings of International Conference on Theory and Practice of Model Transformations, pp. 121–137. Springer (2017). DOI 10.1007/978-3-319-61473-1_9
43. Strüber, D., Born, K., Gill, K.D., Groner, R., Kehrer, T., Ohrndorf, M., Tichy, M.: Henshin: A usability-focused framework for EMF model transformation development. In: Proceedings of International Conference on Graph Transformation, pp. 196–208. Springer (2017). DOI 10.1007/978-3-319-61470-0_12
44. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of Higher-Order model transformations. In: Proceedings of European Conference on Model Driven Architecture – Foundations and Applications, pp. 18–33. Springer (2009). DOI 10.1007/978-3-642-02674-4_3
45. Zhang, Y., Harman, M., Mansouri, S.A.: The Multi-objective Next Release Problem. In: Proceedings of Annual Conference on Genetic and Evolutionary Computation, pp. 1129–1137 (2007). DOI 10.1145/1276958.1277179
46. Zitzler, E., Thiele, L.: Multiobjective optimization using evolutionary algorithms - a comparative case study. In: A.E. Eiben, T. Bäck, M. Schoenauer, H.P. Schwefel (eds.) Parallel Problem Solving from Nature - PPSN V, pp. 292–301. Springer (1998)
47. Zschaler, S., Mandow, L.: Towards model-based optimisation: Using domain knowledge explicitly. In: Proceedings of Workshop on Model-Driven Engineering, Logic and Optimization, pp. 317–329 (2016). DOI 10.1007/978-3-319-50230-4_24