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

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

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

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

Abstract—Recently there has been increased interest in combining the fields of 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 consistency preserving search operators (aCPSOs) 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.

Index Terms—model driven engineering, search based optimisation, search based software engineering

I. INTRODUCTION

Search-based software engineering (SBSE) [1] 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, typically 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 [2], making the benefits of domain-specific modelling languages (DSMLs) available in an SBSE context.

Typical approaches (*e.g.,* [3], [4]) use evolutionary algorithms. Users provide small endogenous model transformations (*e.g.,* expressed as Henshin rules [5]) 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. In particular, we generate operators that are consistency preserving, a key property for enabling the search to move out of local optima. We call such operators *consistency preserving search operators* (CPSOs).

We will show, through case-study–based experimental evaluation, that our automatically generated CPSOs 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, 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 [6] 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 generalises the work in [7], where the authors explored initial ideas for rule generation in the context of a single case study without a generalised approach. Specifically, this paper makes the following contributions:

1) A general description and classification of CPSOs;
2) An algorithm for generating atomic CPSOs (aCPSOs) that preserve multiplicity constraints; and
3) An experimental evaluation with 3 case studies.

An extended version of this paper including more detailed descriptions of the proposed generation algorithm and results, has been published separately [8].

The remainder of this paper is structured as follows: In Sect. II we introduce some relevant background, followed by a running example in Sect. III. Section IV contains the main contributions, describing CPSOs and the generation algorithm. Section V presents the experimental setup, followed by Sect. VI in which we discuss results. In Sect. VII we evaluate related work.

II. Background

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

a) Model-driven engineering: MDE considers models to be the primary artefact in software development [9]. 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 [5] is one example, based on graph-transformation theory.

b) Search-based model engineering: Search-based approaches in software engineering often use evolutionary search techniques. Evolutionary search (ES) [10] 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) [10], 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.

Evolutionary algorithms have been applied to MDE in two ways [2], [11]: some approaches (*e.g.,* [3], [4]) encode candidate solutions as transformation chains and apply genetic algorithms to solve the search problems. Other approaches (*e.g.,* [12]) 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.

c) Higher-order transformations: The term higher-order transformations (HOTs) [13] 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 [14], 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 [15, 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 in order to make it consistent with its metamodel. In [16], 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 [16].

d) 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. The optimisation algorithms supported by the tool are implemented using MOEAFramework [2].

The following elements describe an optimisation problem:

- A problem metamodel describing the structure of problems and solutions;
- 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;
- A set of endogenous model transformations typed over the problem metamodel, called mutation operators;
- 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;

Based on these inputs, MDEO runs an ES. The input model is used as a seed for the initial population generation. The tool uses the specified mutation operators to generate new candidate solutions in each algorithm step. Candidate solutions are evaluated after each generation, using the specified constraint and objective functions.

III. 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 [17].

---

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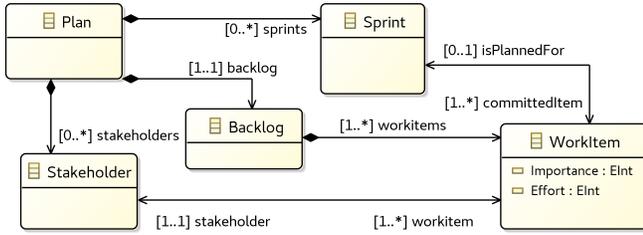
[2] https://moeaframework.org

Fig. 1: Metamodel for the Scrum Planning problem

We will briefly introduce the core Scrum concepts as described in [17]. 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 (SSI) and we calculate it as the standard deviation of average stakeholder importance across sprints.

In Fig. 1 we show a metamodel of this problem. The goal of the problem is to assign `WorkItem (WI)` elements to a number of `Sprints` with the following objectives: **Objective 1** minimise the `Sprint` effort deviation; **Objective 2** minimise the SSI. The problem has the following constraints: **Constraint 1** all `WI` entities must be assigned to a `Sprint`; **Constraint 2** no solution must have fewer `Sprints` than total backlog effort divided by team velocity.

To explore the search space of the Scrum Planning problem, the mutation operators must create `Sprint` entities and assign `WI` elements to them, until all the `WI` elements belong to a `Sprint`. The solution constraints include a refined multiplicity (with a lower bound of 1) for the *sprints* edge between a `Plan` and `Sprint` and also for the *isPlannedFor* edge between a `WI` and a `Sprint`.

IV. 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 `WIs` to them.

A. 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 `WI` from one `Sprint` to another. Once all the `WI` elements have been assigned to a `Sprint`, no more new `Sprint` nodes can be created: because there are no more free `WI` elements, the lower-bound constraint that no `Sprint` should be empty can no longer be satisfied for these new `Sprints`. If all the `WI` 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 `WIs` across the created `Sprints`. Note that creating two mutation operators, one to create an empty `Sprint` and one to move an existing `WI` 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 `WI` that has not yet been assigned to another `Sprint` and assign it to the new `Sprint`. In the second phase, this rule is not applicable anymore, because no unassigned `WIs` remain.
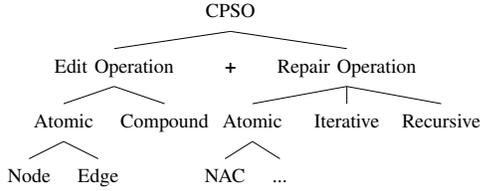
Fig. 2: CPSOs structure

However, there is an alternative repair which takes a `WI` from an existing `Sprint` with at least two `WI` elements assigned to it. We need to generate appropriate mutation operators for each phase of the search.

Mutation operators that satisfy these requirements, we will call Consistency-Preserving Search Operators (CPSOs).

B. General structure of CPSOs

As we have seen in the previous sub-section, CPSOs are transformation rules that combine a change to the model (an edit operation) with the necessary repair. In Fig. 2 we show the structure of CPSOs 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 consistency preserving search operators (aCPSOs).

C. 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 [14], [15], 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 CPSOs 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 aCP-SOs. 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,

TABLE I: Create node aCPSOs. 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

remove, change, and swap. The available repair operations depend on the multiplicity pattern. Fig. 3 shows the labels we will use in our discussion below.

Fig. 3: Multiplicity patterns

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. The minimal set of rules, ensures that we can perform all the create and delete node operations and add and remove edges between graph elements.

1) Manipulating nodes: In this section we describe the repair operations required for manipulating nodes. The types of aCPSOs 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. Kehrer et al. support a restricted set of multiplicities for creating and deleting a node of type A: (k,l) to (0,1) or (k, l) to (0,*).

a) Creating a node: In this section we introduce the types of repair operations generated for creating a node. 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 I.

- *NAC repair*: The first type of aCPSO 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.
- *Single source lower bound repair*: The second type of aCPSO 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, we generate a repair to satisfy the lower-bound

TABLE II: Delete Node aCPSOs. 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

for the created node A and 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 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 node A to ensure that the lower-bound multiplicity is not broken by this operation.

- *Multiple sources lower bound repair*: The third type of aCPSO 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 generate a repair to satisfy the lower-bound for the created node A and 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.

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

b) 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 II.

- *PAC repair*: The first type of aCPSO that we generate, is for deleting nodes that have a closed multiplicity $(k > 0)$. 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 $(k = l)$.

- *Single target lower bound repair*: This type of aCPSO for deleting a node, is for deleting nodes that have a multiplicity pattern with $(k = 1)$. This pattern means that each node of type B must be assigned to at least *k* nodes of type A. For this case, we generate a repair to satisfy the lower-bound for the k nodes B, by creating edges between them and another single existing node of type A. A NAC

TABLE III: Add-edge aCPSOs. 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 IV: Remove-edge aCPSOs. 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

is generated for the existing node A to ensure that the upper-bound multiplicity is not broken if $(m < *)$.

- *Multiple target lower bound repair delete*: This type of aCPSO for deleting a node, is 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 at least *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.

For node pairs that have a fixed multiplicity $(n = m \land k = l)$, at both ends of any edge, we do not generate a delete node aCPSO. 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.

2) Manipulating edges: In this section we show the types of aCPSOs we generate for manipulating edges between two nodes. Namely to add and to 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 aCPSOs is included in Tables III and IV.

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

b) Removing an edge: This aCPSO is also similar 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 aCPSO includes a NAC to avoid invalidating any lower-bound constraints between the source and target nodes. This aCPSO is generated for all multiplicity patterns except for cases having a fixed multiplicity at one end or at both ends of the connected nodes ($n = m \lor k = l$).

c) Changing an edge: A change edge aCPSO moves a node of type B to another node of type A when node B has a fixed multiplicity. This type of operation, moves a node of type B with a lower bound multiplicity pattern ($k > 0$), to another node of type A, without invalidating the multiplicity constraints. The generated aCPSO 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 to few nodes of type B after this rule application). This aCPSO is generated for closed multiplicity patterns where a multiplicity pattern for either of the connected nodes is fixed (*e.g.,* $n = m \vee k = l$).

d) Swapping two edges: An edge swap aCPSO 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 aCPSO, 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.

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.

D. Running search with aCPSOs

The algorithm proposed in the previous sub-section generates operators that preserve the consistency of the models modified. This addresses the first requirement on mutation operators that we identified in Sect. IV-A. 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. IV-C 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.

V. EXPERIMENTS

To evaluate our rule generation approach, we ran a number of experiments with a range of case studies. The aim of these experiments is 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 show that there is no loss in search performance from generated operators. For each case study, we prepared a set of manually created mutation operators. Then, we configured MDEO to automatically generate mutation operators. Using both pairs of mutation operators we ran experiments to solve

the same problem instances, and we compare the results from the two approaches to validate that the solutions obtained using the automatically generated aCPSOs are comparable with the results obtained using manually implemented search operators.

We are not including a comparison between our tool and other tools. Such a comparison is beyond the scope of this paper. In [11] the authors compare the performance of MDEO and MoMOT, another model search tool that encodes search solutions as transformation chains.

A. Case studies

We have chosen a set of combinatorial optimisation problems that have been implemented using MDEO. In the following subsections, we include a description of each case study.

1) Class-Responsibility Assignment: The Class Responsibility Assignment (CRA) case study has been introduced at the 2016 Transformation Tool Contest (TTC) [18]. 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 [18], 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. Each model differs in the number of Methods and Attributes present. A summary of all the input models is included in Table V.

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 [19]. Other TTC'16 participants that used a similar

TABLE V: Summary of CRA input models

	A	B	C	D	E
Attributes	5	10	20	40	80
Methods	4	8	15	40	80
Data Dep.	8	15	50	150	300
Functional Dep.	6	15	50	150	300

approach to solve the case studies used similar rules [18], [20]. The second set of operators are aCPSOs generated using the approach presented in this paper.

2) Scrum Planning: We are running two experiments for the Scrum Planning (SP) case study described in Sect. III. 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 VI 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

TABLE VI: Summary of input models for SP and NRP case studies.

Scrum Planning			Next Release Problem		
Input Model	A	B	Input Model	A	B
Stakeholders	5	10	Customers	5	25
WorkItems	119	254	Requirements	25	50
Backlog Size	455	1021	Software Artifacts	63	203

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.

3) Next Release Problem: The aim 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 [21]. Each `Customer` has a desire which can consist of one or many `SoftwareArtifacts` (SA). SAs can have a recursive dependency on other SAs.

To solve this problem, the user is required to assign instances of SAs to a `NextRelease` such that the total cost of the selected SAs 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 CPSOs, 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 `SA`. However, the difference between this case study and the others considered in this paper is that the selection of a `SA`, directly influences the Cost fitness function and indirectly influences the Customer Satisfaction objective. A SA is considered for the calculation of a `RequirementRealization`, only when all its dependencies are also assigned to the solution. 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 VI.

B. 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.

a) Experimental Setup: All the experiments have been repeated 30 times for statistical significance [22] and have been 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 [23]. The

NSGA-II algorithm is a well established algorithm that has been used successfully in many SBSE applications [2].

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 was for comparing the quality of GEN with MAN.

b) Hyperparameter Selection: 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 have each been repeated 10 times to ensure robust results.

Because of space limitations, the results of the parameter searching configurations have been included in the data attachment for this paper. Upon evaluating the results across all case studies, we determined the population size of 100 to be the best value for our experiments. The selected number-of-evolutions parameter values have been included in the Evol column in Table VIII for the CRA case, Table X for the SP case and Table XII for the NRP case.

c) 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 HV indicator has been proposed in [24]. 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 HVs indicate a Pareto front closer to the theoretical optimum.

d) 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 [25]. 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.

e) Statistical Analysis: We use the Mann-Whitney U test to perform a statistical analysis of our results [26]. To measure the size of the differences between the configurations we use Cohen's d effect metric [27]. We are also including 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.

VI. 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 discussed in this section can be downloaded from [28].

A. Class Responsibility Assignment

Table VII shows the mutation operators used for the two experiment configurations. Both configurations use three identical mutation operators to create a class (Create Class) and assign and change a feature (Assign Feature, Change Feature).

TABLE VII: CRA mutation operators for MAN and GEN.

Manual	Gen aCPSO
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 VIII: CRA results for MAN and GEN.

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 IX: 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

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 operator to change a `Feature` 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`. The performance of the GEN operators is not affected.

Table VIII shows summary statistics for the CRA index found using the two configurations. 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 IX 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 aCPSO 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 show that our approach is good at generating mutation operators that help find results that are comparable to manually specified mutation operators.

B. Scrum Planning

The SP case study is specified as a multi-objective problem. To compare the results we will use the HV metric. In Table XI we include the mutation operators used for the two experiment configurations. These mutation operators are similar to the ones generated for the CRA case study.
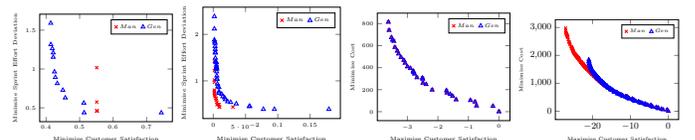
TABLE X: SP results for Man and Gen configurations.

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

TABLE XI: SP mutation operators for MAN and GEN.

Manual	Gen aCPSO
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

Table. X shows a comparison of the calculated HVs 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 HV, because there are no constraint satisfying solutions generated and the covered HV 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 HV values for the configurations with valid solutions, we observe that GEN is better than MAN for both input models.

For model A, the highest HV 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. 4a 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 HV 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. 4b 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.

(a) SP A	(b) SP B	(c) NRP A	(d) NRP B

Fig. 4: Pareto fronts for SP and NRP cases.

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

TABLE XIII: NRP mutation operators for MAN and GEN.

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

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 is able to 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 `WI` elements between `Sprints`, until the right configuration is found.

For this case study, GEN finds a consistently good HV, with a small SD value across all the repetitions, as seen in the Median and SD columns in Table. X. The difference between the numbers of valid solutions found, shows that the addition of the aCPSOs helped the search to find consistent solutions.

C. Next Release Problem

In contrast to the other use cases, the operators used in the NRP case are substantially different for both configurations (Table XIII). The GEN operators only cover the basics: addition and removal of single `SA`s. In both 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 `SA` 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 `SA`s 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 fulfillment and adds its missing `SA`s 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 `SA` or removing the dependent artifacts of a formerly removed `SA`.

For model A, we can observe that GEN consistently finds a HV 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 HVs across all 30 runs for each configuration is 0. This can also be observed in Fig. 4c 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, however these can be found in the data attachment. For model B, the HV 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. 4d, 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 XII. 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 the way in which the MAN operators have been designed, compared to the ones automatically generated. The MAN operators, ensure that a `SA`, together with all its dependencies are all assigned to a Solution in a single application. At the same time, the GEN operators, assign `SA`s, one by one, by adding or removing edges, in atomic operations. Because the Customer Satisfaction objective does not provide guidance for the solutions, unless all `SA`s realising a complete Realization 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 `SA` together with all its dependencies in a single evolution, is equivalent to running the GEN configuration operator to add an edge, for multiple evolutions. We are currently investigating how different strategies for adjusting the step size can help aCPSOs overcome this issue and the initial results show that with the correct step size selection strategy the search algorithm can overcome limitations similar to the one observed in this case.

D. Performance

We have also compared the efficiency of the generated operators with the manually created ones. In all cases but one, the generated operators led to shorter (or at most equal) average runtimes for the ES. For two scenarios the generated rules were less efficient than the manual ones. This happened for the CRA and SP case studies. For CRA we additionally explored two different matching strategies in MDEOptimiser: the *classic strategy*[3] 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,

---
[3]Which the tool authors used in their submission to TTC'16 [19]

under this strategy the search with generated rules took up to approximately 3 times as long as with manual rules. With the 'non-deterministic matching strategy', the generated rules led to a faster search than the manual rules. For SP, GEN finds more constraint satisfying solutions and more time is spent evaluating the fitness functions. This leads to an increase in the runtime for `Gen`. The runtimes for both configurations for all case studies have been included in the data attachment accompanying the paper.

E. 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 representative of real-world problems, and 4) the provenance of the manual rules used in our experiments.

We have 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 overal problem. All hyperparameter values have been selected systematically to ensure that no approach is favoured over the other. We have applied the recommended steps to ensure that our results are accurate and correctly interpreted and described [22]. Input models have been either provided as part of pre-existing case studies (CRA [18]), or have been randomly generated, ensuring consistency with the given problem metamodel. Recently, better model generators have been proposed [29], [30] that aim to produce more realistic model instances for such evaluations. We are interested to explore the use of such generators for further evaluation of our approach. Finally, the manual rules that we use 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 [19], well before we started considering automatic generation of rules; the SP rules are very similar to the CRA rules. The NRP rules were produced by the 3rd author taking into account the structure of the objective functions during rule construction.

VII. RELATED WORK

a) 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 [6] 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 [31]. 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.

In [32], the authors present an offline hyper-heuristic approach that automatically generates mutation operators using genetic programming and meta-learning. These are then used in evolutionary programming with the aim of solving a number of 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 DSML. The work in [32] is similar to [6], 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.

In [33] the authors 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 languages, 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. in [34] propose a complete DSL operator library for EMF based languages. The operators are atomic operators and the proposed list of the most used operators also contains the aCPSO and CPSO 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.

b) Mutation Weighting: In [35], the authors 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 a better fitness. Our case study evaluation showed that this is also a problem affecting our approach. For NRP and the case study presented in [36], the generated aCPSOs 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 aCPSO application. Alternatively, the problem can be solved using a combination of operators consisting of aCPSOs and compositions of multiple aCPSOs that are applied in a single step. This is a problem we seek to explore in future work to expand our mutation generation approach.

VIII. Conclusions

We have shown 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 consistency-preserving search operators (aCPSOs) we generate is 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:

- 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 CPSOs, 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?

Acknowledgment

References

[1] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[2] I. Boussaïd *et al.*, "A survey on search-based model-driven engineering," *Automated Software Engineering*, vol. 24, pp. 233–294, Jun 2017.

[3] M. Fleck, , *et al.*, "Search-based model transformations," *Journal of Software: Evolution and Process*, vol. 28, no. 12, pp. 1081–1117, 2016.

[4] Á. Hegedüs *et al.*, "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, Nov. 2011.

[5] Strüber *et al.*, "Henshin: A usability-focused framework for EMF model transformation development," in *Int. Conference on Graph Transformation*, pp. 196–208, Springer, 2017.

[6] D. Strüber, "Generating efficient mutation operators for search-based model-driven engineering," in *Int. Conference on Theory and Practice of Model Transformations*, pp. 121–137, 2017.

[7] A. Burdusel and S. Zschaler, "Towards automatic generation of evolution rules for model-driven optimisation," in *8th International Workshop on Graph Computation Models*, pp. 60–75, 2018.

[8] A. Burdusel, S. Zschaler, and S. John, "Automatic generation of atomic consistency preserving search operators for search-based model engineering." http://dx.doi.org/10.6084/m9.figshare.8863640.

[9] J. Bézivin, *Model Driven Engineering: An Emerging Technical Space*, pp. 36–64. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.

[10] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Springer Publishing Company, Incorporated, 2nd ed., 2015.

[11] S. John *et al.*, "Searching for optimal models: Comparing two encoding approaches," *Journal of Object Technology*, vol. 18, no. 3, 2019.

[12] S. Zschaler and L. Mandow, "Towards model-based optimisation: Using domain knowledge explicitly," in *Proc. Workshop on Model-Driven Engineering, Logic and Optimization (MELO'16)*, 2016.

[13] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, "On the use of higher-order model transformations," in *Proc. 5th European Conf. on Model Driven Architecture – Foundations and Applications (ECMDA-FA'09)* (R. F. Paige, A. Hartman, and A. Rensink, eds.), vol. 5562 of *Lecture Notes in Computer Science*, pp. 18–33, Springer, 2009.

[14] T. Kehrer, G. Taentzer, M. Rindt, and U. Kelter, "Automatically deriving the specification of model editing operations from meta-models," in *Int. Conference on Theory and Practice of Model Transformations*, 2016.

[15] T. Kehrer, *Calculation and propagation of model changes based on user-level edit operations: a foundation for version and variant management in model-driven engineering*. PhD thesis, University of Siegen, 2015.

[16] N. Nassar, H. Radke, and T. Arendt, "Rule-based repair of emf models: An automated interactive approach," in *Int. Conference on Theory and Practice of Model Transformations*, pp. 171–181, Springer, 2017.

[17] K. S. Rubin, *Essential Scrum*. Addison-Wesley, 2012.

[18] M. Fleck, J. Troya, and M. Wimmer, "The class responsibility assignment case," in Garcia-Dominguez *et al.* [37], pp. 1–8.

[19] A. Burdusel and S. Zschaler, "Model optimisation for feature class allocation using MDEOptimiser: A TTC 2016 submission," in Garcia-Dominguez *et al.* [37], pp. 33–38.

[20] A. S. Nagy and G. Szárnyas, "Class responsibility assignment case: a viatra-dse solution," [37], pp. 39–344.

[21] Y. Zhang *et al.*, "The multi-objective next release problem," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, (New York, NY, USA), pp. 1129–1137, ACM, 2007.

[22] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[23] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[24] E. Zitzler *et al.*, "The hypervolume indicator revisited: On the design of pareto-compliant indicators via weighted integration," in *Evolutionary multi-criterion optimization*, pp. 862–876, Springer, 2007.

[25] M. Hansen and A. Jaszkiewicz, "Evaluating the quality of approximations to the non-dominated set," Tech. Rep. IMM-REP-1998-7, Institute of Computing Science, Poznan University of Technology, 03 1998.

[26] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Statist.*, vol. 18, pp. 50–60, 03 1947.

[27] J. Cohen, "Statistical power analysis for the behaviors science.(2nd)," *New Jersey: Laurence Erlbaum Associates, Publishers, Hillsdale*, 1988.

[28] A. Burdusel, S. Zschaler, and S. John, "Automatic generation of atomic consistency preserving search operators for search-based model engineering data." http://dx.doi.org/10.6084/m9.figshare.8316740.

[29] O. Semeráth, A. S. Nagy, and D. Varró, "A graph solver for the automated generation of consistent domain-specific models," in *Int. Conference on Software Engineering*, pp. 969–980, ACM, 2018.

[30] S. Schneider, L. Lambers, and F. Orejas, "Symbolic model generation for graph properties," in *Int. Conference on Fundamental Approaches to Software Engineering*, pp. 226–243, Springer, 2017.

[31] M. Fleck, J. Troya, and M. Wimmer, "The Class Responsibility Assignment Case," in Garcia-Dominguez *et al.* [37].

[32] L. Hong, J. H. Drake, J. R. Woodward, and E. Özcan, "A hyper-heuristic approach to automated generation of mutation operators for evolutionary programming," *Applied Soft Computing*, vol. 62, pp. 162 – 175, 2018.

[33] F. H. M. Alhwikem *et al.*, "A systematic approach for designing mutation operators for mde languages," in *Proceedings of the 13th Workshop on Model-Driven Engineering, Verification and Validation co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)*, 2016.

[34] J. G. M. Mengerink *et al.*, "A complete operator library for dsl evolution specification," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 144–154, Oct 2016.

[35] B. Doerr, C. Doerr, and T. Kötzing, "The right mutation strength for multi-valued decision variables," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1115–1122, ACM, 2016.

[36] J. Murphy *et al.*, "Deriving persuasion strategies using search-based model engineering," in *Computational Models of Argument: Proceedings of COMMA 2018*, vol. 305, pp. 221–232, IOS Press, 2018.

[37] A. Garcia-Dominguez, F. Krikava, and L. M. Rose, eds., *Proceedings of the 9th Transformation Tool Contest*, vol. 1758, CEUR, 2016.