A local and global tour on MOMoT

Robert Bill[1] · Martin Fleck[2] · Javier Troya[3] · Tanja Mayerhofer[1] · Manuel Wimmer[4]

Abstract
Many model transformation scenarios require flexible execution strategies as they should produce models with the highest possible quality. At the same time, transformation problems often span a very large search space with respect to possible transformation results. Recently, different proposals for finding good transformation results without enumerating the complete search space have been proposed by using meta-heuristic search algorithms. However, determining the impact of the different kinds of search algorithms, such as local search or global search, on the transformation results is still an open research topic. In this paper, we present an extension to MOMoT, which is a search-based model transformation tool, for supporting not only global searchers for model transformation orchestrations, but also local ones. This leads to a model transformation framework that allows as the first of its kind multi-objective local and global search. By this, the advantages and disadvantages of global and local search for model transformation orchestration can be evaluated. This is done in a case-study-based evaluation, which compares different performance aspects of the local- and global-search algorithms available in MOMoT. Several interesting conclusions have been drawn from the evaluation: (1) local-search algorithms perform reasonable well with respect to both the search exploration and the execution time for small input models, (2) for bigger input models, their execution time can be similar to those of global-search algorithms, but global-search algorithms tend to outperform local-search algorithms in terms of search exploration, (3) evolutionary algorithms show limitations in situations where single changes of the solution can have a significant impact on the solution's fitness.

Keywords Model-driven engineering · Model transformation · Search-based software engineering · Local search · Global search

✉ Javier Troya
jtroya@us.es

Robert Bill
bill@big.tuwien.ac.at

Martin Fleck
mfleck@eclipsesource.com

Tanja Mayerhofer
mayerhofer@big.tuwien.ac.at

Manuel Wimmer
wimmer@big.tuwien.ac.at

1 Business Informatics Group, TU Wien, Vienna, Austria

2 EclipseSource Services GmbH, Vienna, Austria

3 Department of Computing Languages and Systems, Universidad de Sevilla, Sevilla, Spain

4 CDL-MINT, TU Wien, Vienna, Austria

1 Introduction

Model transformations are the key technology to manipulate models in model-driven engineering (MDE) [9]. As the applicability of MDE is expanding in software engineering and beyond, model transformations have to cope with many new challenges. One of these challenges is how to deal with the large search spaces of many transformation problems. Of course, one approach is to develop problem-specific heuristics that allow to deal with the associated search space without having to enumerate all possible solutions, which is mostly not possible due to practical space and time restrictions. However, finding such problem-specific heuristics is challenging. Therefore, an alternative approach is the usage of meta-heuristics that are problem independent. This is investigated by Search-Based Software Engineering (SBSE) [35], which is a lively research field applying search-based optimization techniques to software engineering problems. Search-based optimization techniques deal with large or even infinite search spaces in an efficient manner. Concrete algorithms

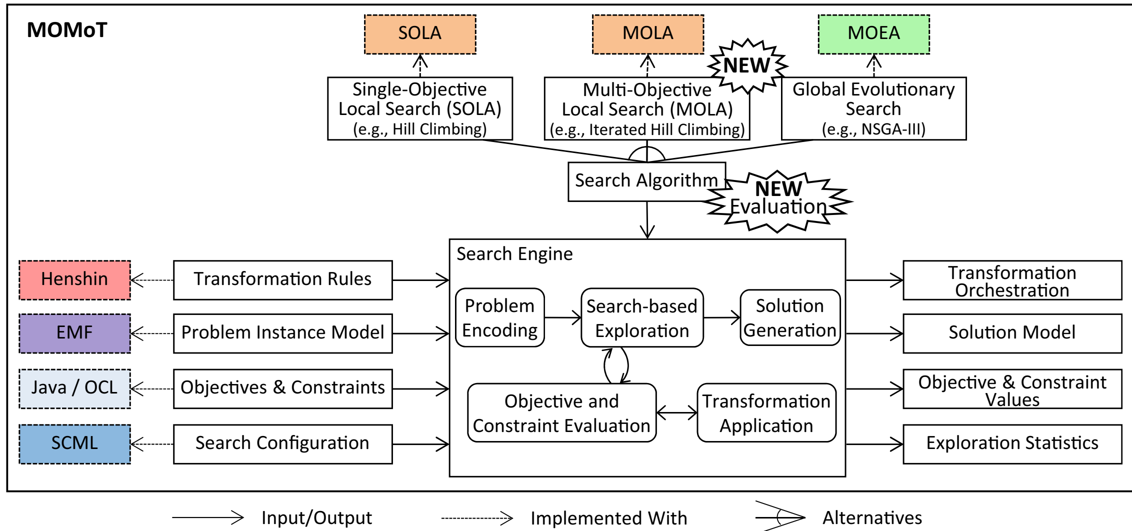

Fig. 1 Overview of MOMoT

include local-search methods, such as Tabu Search [32] and Simulated Annealing [46], or genetic algorithms [39], such as NSGA-II [16] and NSGA-III [15].

In recent years, SBSE has been applied successfully in the area of MDE [45]. Examples include the generation of model transformations from examples [41,68,71], the optimization of regression tests for model transformations [63], the detection of high-level model changes [6], and the enhancement of the readability of source code for given metrics [22,23]. Very recently, several approaches have been proposed to provide more efficient search capabilities for model transformations [1,17,25].

MOMoT is one of these emerging approaches and was first presented in [25]. It relies on Henshin [4] as base model transformation framework and MOEA[1] as base meta-heuristic search framework. Henshin is a graph-based model transformation framework that offers a rule-based language and associated tool set for in-place model-to-model transformations. The MOEA framework is an open-source Java library that provides a set of multi-objective evolutionary algorithms with additional analytical performance measures and that can be easily extended with new algorithms. Thus, MOMoT combines different search techniques with model transformations to produce output models that optimize one or more potentially conflicting quality criteria. Reusing the existing functionality of these base frameworks as much as possible is the central principle of our framework. While in the rest of the paper we discuss our framework in light of Henshin and MOEA, the conceptual approach itself is generic so that it may be used for other framework combinations. Please note, however, that MOMoT is for now focused on in-place model transformations. Considering the application of search-based optimization techniques on out-place model transformations is out of the scope of this paper and subject to future work.

An overview of MOMoT is shown in Fig. 1. Like typically done in MDE, the studied problem domain is defined with a meta-model using the meta-modeling language Ecore offered by the Eclipse Modeling Framework (EMF) [64]. The concrete problem can then be defined with the help of an instance of this meta-model referred to as problem instance model. The model transformation that should transform the problem instance model into a solution model that fulfills certain quality criteria is defined with the model transformation language Henshin. Using Java or OCL, the quality criteria of the desired solution model are defined comprising objectives and constraints. MOMoT's search engine is then responsible for investigating the search space to find an orchestration of the model transformation rules that produces a good solution model with regard to the defined quality criteria. For investigating the search space, MOMoT provides different search algorithms including global evolutionary search algorithms offered out-of-the-box by MOEA and single-objective local-search algorithms (SOLA) implemented in our previous work [25]. The output provided by MOMoT does not only comprise the found model transformation rule orchestration and solution model, but also the values computed for the defined objectives and constraints for the found solution, as well as statistical information about the performed search.

A novel feature of MOMoT and a contribution to the model transformation literature of this paper is the application of multi-objective local-search approaches (MOLA) for model transformations. We are interested in this kind of search approach as model transformations are naturally considered as multi-objective problems. Even if only one

[1] http://www.moeaframework.org.

particular objective should be minimized or maximized in the output model of the transformation, normally one would be in addition interested in finding a short transformation sequence to reach this result, thus turning the problem in a multi-objective problem [1]. In this paper, we contribute different multi-objective local-search approaches to MOMoT, which have not been supported by MOMoT before, with the aim of improving MOMoT's capability to find good model transformation orchestrations.

In fact, the existence of both local-search approaches and global-search approaches is leading directly to the question, which search approach is the best for a certain application domain, such as model transformations in our case. Currently, there is no empirical work on comparing these two different approaches for model transformations despite the fact that both approaches have been proposed for solving model transformation problems [1,17,25,28,29]. In this paper, we investigate this question by comparing a set of local-search approaches and a set of global-search approaches for three case studies taken from different domains. In particular, we evaluate different performance aspects, such as the quality of the search exploration, i.e., how much of the search space is actually covered and how good the found results are, and the time needed for this exploration.

Please note that this article is an extension of our previous publications [26,28]. In [28], we have presented the overall MOMoT framework with its support of (standard single-objective) local-search algorithms and global evolutionary search algorithms. The evaluation of MOMoT presented in [28] focused on validating its applicability to model-based software engineering problems and the runtime overhead introduced by following a model-based approach compared to a native encoding. In this evaluation, only the global evolutionary search Algorithm NSGA-III was considered. In [26], we have presented the integration of Henshin and MOEA from an architectural point of view and demonstrated the concrete tool support for specifying search-based model transformations by using the Search Configuration Modeling Language (SCML). Besides incremental advancements of the general model transformation approach, this article introduces two major extensions over the previous publications: First, we have extended MOMoT to also support multi-objective local-search approaches to orchestrate model transformation rules in addition to the global evolutionary algorithms. Second, apart from this technical contribution, we have significantly improved the evaluation of our approach. In particular, we now compare the different meta-heuristic search approaches concerning performance in three case studies of varying complexity and subject. To the best of our knowledge, this is the first evaluation of this kind comparing the performance of local-search approaches and global-search approaches for their application to model transformations.

The remainder of this paper is structured as follows. First, we introduce the background for this paper in Sect. 2, which is mostly concerned with explaining meta-heuristic search and its application for model transformations, as well as presenting the running example for this paper. Section 3 describes MOMoT and demonstrates its capabilities on the running example. Section 4 then introduces our extensions of MOMoT and MOEA with multi-objective local-search algorithms. Section 5 is dedicated to evaluating the different search algorithms offered by MOMoT by using a case-study-based evaluation approach. Finally, Sect. 6 discusses related work and we conclude the paper with an outlook on future work in Sect. 7.

2 Preliminaries: meta-heuristics and model transformations

In this section, we describe meta-heuristic search, model transformations and their joint application. Furthermore, a motivating example is introduced, which is later on used as a running example for the rest of this paper.

2.1 Meta-heuristic search

In this section, we discuss how meta-heuristic search is used for turning engineering problems into optimization problems, and subsequently show which different search approaches are available for solving such problems.

2.1.1 Reformulating engineering problems as optimization problems

A meta-heuristic is a heuristic method aimed at resolving a general computation problem. In particular, it is applied on problems that do not have a specific algorithm or heuristic that produces a satisfactory solution, or when it is not possible to implement such optimal method. Most meta-heuristics target the resolution of problems of combinatorial optimization, but they can be applied to any problem that can be reformulated in heuristic terms.

Search-based software engineering [35] is a field that applies search-based optimization techniques to software engineering problems. Thereby, search-based optimization techniques can be categorized as meta-heuristic approaches that deal with large or even infinite *search spaces* in an efficient manner. Therefore, SBSE techniques are often applied on problems where it is not feasible to use exact or enumerative approaches [74].

Formally, a solution to a given optimization problem can be seen as a vector of decision variables in the decision space X. In order to evaluate the quality of a solution, a fitness function $f : X \mapsto Z$ maps a given solution to an objective

vector in the objective space Z. A meta-heuristic approach uses these mappings to manipulate the decision variables of a solution in such a way that we reach good values in the objective space. The notion of good values relates to the direction of the optimization; typically, an objective value in the objective vector needs to be minimized or maximized. Additionally, a solution may be subject to a set of inequality constraints denoted $g_i(x) \geq 0$ with $i = 1, \ldots, P$ and a set of equality constraints denoted $h_j(x) = 0$ where $j = 1, \ldots, Q$. A solution satisfying the $(P + Q)$ constraints is said to be feasible, and the set of all feasible solutions defines the feasible search space.

Meta-heuristic approaches dealing with large or infinite search spaces are often divided in two groups, namely local-search methods and global-search algorithms, commonly referred to as evolutionary algorithms. The aim of local-search methods is to improve one single solution at a time, while evolutionary algorithms [39] manage a set of solutions, called a population, at once. The number of objectives that need to be optimized, i.e., the size of the objective vector, is also used to categorize optimization problems. Single-objective or mono-objective problems only deal with one objective at a time, multi-objective problems deal with more than one objective. Recently, due to the limits of how many objectives different algorithms can handle, a distinction is made between multi-objective problems and many-objective problems. A many-objective problem has at least four objectives, while multi-objective problems are known to address problems with two and three objectives.

2.1.2 Local-search algorithms

Local-search methods start with an initial solution and then perform two steps in each iteration: generation and moving. The local-search process is shown in Algorithm 1 [65] and visually depicted in Fig. 2. In the first step, a set of candidate solutions called a neighborhood is generated from the current solution. In the second step, a solution from the generated neighborhood is selected to replace the current solution. The whole process iterates until the given stopping criterion is fulfilled, e.g., a specific solution is found, a solution with sufficient quality is found or the algorithm has run a given number of iterations.

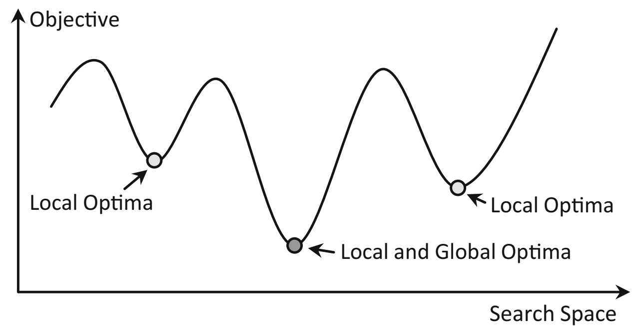

Fig. 3 Local optimum and global optimum in a search space (from [65])

Algorithm 1: High-level Process of Local Search [65]

Input: Initial Solution s_0
Output: Best Solution Found

```
1  t = 0
2  repeat
3      Generate(N(s_t))
       // Generate partial or complete neighborhood
4      s_{t+1} = Select(N(s_t))
       // Select a neighbor to replace current
          solution
5      t = t + 1
6  until Stopping criteria satisfied
```

In local search, neighbors consist of solutions that represent a small move from the current solution, i.e., a small change in the representation of the current solution. A key challenge of local-search algorithms is to not get stuck in low-quality local optima. The relation between local and global optima is depicted in the fitness landscape of Fig. 3 [65] for a minimization problem. In a search space, many neighborhoods with different local optima may exist; however, only a subset of these local optima are actually global optima. In many cases, there is even only one globally optimal solution. Knowing the fitness landscape can help to select an algorithm that is able to not get stuck in a local optimum. For instance, in Hill Climbing or Gradient Descent [60], we start with an initial, random solution in the search space and always select a neighbor with a better fitness than the current solution. As such, depending on where in the fitness landscape we start, we may find a local optimum or even a global optimum. However, once a local optimum is found, there is no way of

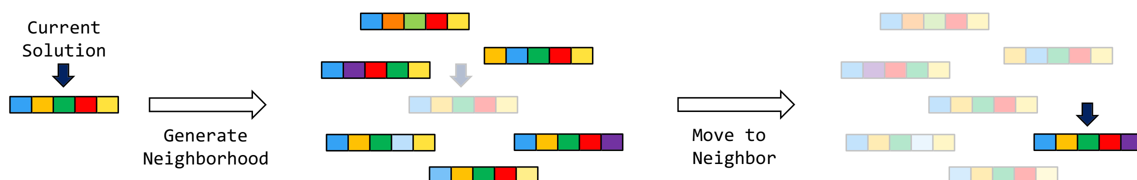

Fig. 2 Two steps performed by local-search methods in each iteration

exploring other parts of the search space and therefore there is no guarantee to find a global optimum.

Many different local-search algorithms have been proposed. These algorithms differ in whether they use a memory or are memoryless (for addressing the problem of finding local optima instead of global optima), in how they generate the initial solution, in how they generate the neighborhood and in how they select the next solution. Therefore, different strategies have been proposed, such as Hill Climbing [11], Random Descent, Tabu Search [32], Simulated Annealing [46] and Gradient Descent [60].

2.1.3 Global-search algorithms

Global-search algorithms, typically referred to as evolutionary algorithms [39], are stochastic, population-based[2] algorithms that build upon the Darwinian principles of evolution [13], i.e., the struggle of individuals to survive in an environment with limited resources and the process of natural selection. In the 1980s, different approaches to incorporate this process into algorithms have been proposed. Examples are Genetic Algorithms [37,38], Evolution Strategies [55, 56], Evolutionary Programming [30,31] and Genetic Programming [49]. Nowadays, some widely used evolutionary algorithms include the Strength Pareto Evolutionary Algorithm 2 (SPEA2) [73], the Non-Dominated Sorting Genetic Algorithm-II [16] (NSGA-II), and the NSGA-III [15]. Evolutionary algorithms are the most studied population-based algorithms [65], and the field of evolutionary multi-objective optimization (EMO) is considered one of the most active research areas in evolutionary computation [14].

Algorithm 2: High-level Process of Global Search [65]

Output: Best Solutions Found
1. $Generate(P(0))$
2. $t = 0$
3. **while** *not termination_criterion(P(t))* **do**
4. $\quad Evaluate(P(t))$
5. $\quad P'(t) = Selection(P(t))$
6. $\quad P'(t) = Reproduction(P'(t))$ `// Recombination and Mutation`
7. $\quad Evaluate(P'(t))$
8. $\quad P(t+1) = Replace(P(t), P'(t))$
9. $\quad t = t + 1$
10. **end**

The high-level process of an evolutionary algorithm is shown in Algorithm 2. In general, evolutionary algorithms start with a population of individual solutions. This initial population is often randomly generated or may be produced in some other form. Every individual in the population is a solution with a specific fitness assigned by the fitness function. In order to manipulate the population toward good areas of the search space, evolutionary algorithms typically use three search operators and a replacement scheme. The algorithm stops when the defined stopping criterion is satisfied, e.g., a specific number of iterations or evaluations have been performed or no improvement has been achieved for a given number of iterations.

The first search operator, called the *selection operator*, chooses individuals from the population and selects them for reproduction. Here, different selection strategies may be applied, e.g., based on absolute or relative fitness of a solution. In any case, the idea is that solutions with a higher fitness are more likely to reproduce. For instance, in the deterministic tournament selection illustrated in Fig. 4, we select k random solutions from the current population and choose the best n solutions for recombination. In non-deterministic tournament selection, the solution with the highest fitness is selected with a given probability p, the next best solution is selected with probability $p * (1 - p)$, the next best with $p * (1 - p)^2$ and so on.

In the *reproduction phase*, variation operators are applied on the selected solutions in order to produce new solutions. The terms parent and child solutions, or parent and offspring populations are used. The most common variation operators are the recombination, or crossover, operator and the mutation operator.

The *recombination operator* is an n-ary operator that takes n parent solutions and produces n child solutions. The idea is that the characteristics that make a solution good can be given to its children, and if two good solutions are (re-)combined, an even better solution may emerge. The most basic recombination strategy is the one-point crossover, where two parent solutions are cut at the same random position and the children are created by a crosswise merge of the resulting parts. This strategy is also depicted in Fig. 4.

The *mutation operator* is an unary operator that introduces small, random changes into a single solution. The main idea behind mutation is that it guides the algorithm into areas of the search space that would not be reached through recombination alone and avoids the convergence of the population toward a few elite solutions.

In the *replacement phase* after selection and reproduction, we need to choose which solutions from the parent population and the offspring population should be kept for the next iteration. The two extreme cases are the replacement of the complete parent population with the offspring population and the replacement of only one individual in the parent population, e.g., the worst individual, through the best individual from the offspring population. Of course, in practice, any number of individuals may be replaced. Another replacement strategy is elitism, i.e., the selection of only the best solutions

[2] Barring some small connotations, for simplicity, we treat the terms global search, evolutionary and population-based algorithms equally.

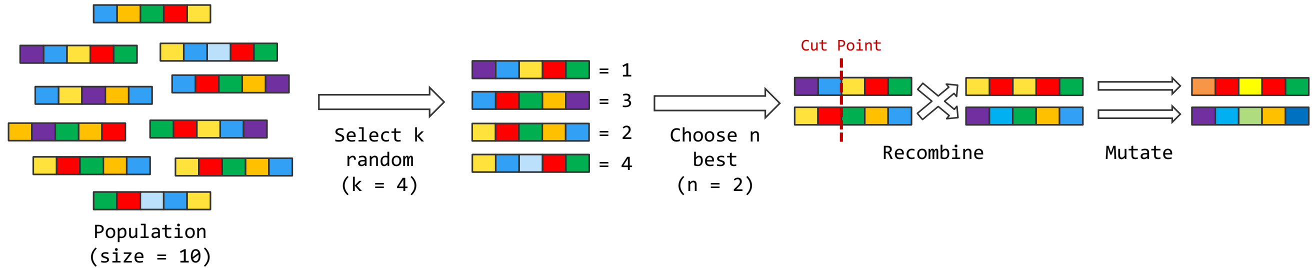

Fig. 4 Steps performed by global-search methods in each iteration

from both populations. This leads to faster convergence, but may result in premature convergence.

2.2 Search-based model transformations

Model transformations are an important cornerstone of model-driven engineering (MDE), a discipline which facilitates the abstraction of relevant information on a problem as models. The success of the outcomes obtained by MDE approaches heavily depends on the optimization of these models through model transformations. When developing a model transformation, a modeler defines rules to manipulate an input model. However, reasoning about how these rules can be applied to retrieve a model with certain characteristics is a non-trivial task. Currently, the application of transformations is realized either by following the apply-as-long-as-possible strategy or explicit rule orchestrations have to be provided. These techniques suffer from several drawbacks, such as the implicitly hidden effect a transformation has on the characteristics of the model, the required knowledge to understand the relationships among transformation rules, i.e., whether they are conflicting or enabling, the large or even infinite number of rule combinations, and the consideration of multiple, potentially conflicting objectives.

In order to address all these difficulties, some works have proposed to apply search-based techniques, such as the recent approaches by Abdeen et al. [1] who integrate multi-objective optimization techniques to drive a rule-based design exploration, and Denil et al. [17] who integrate single-state search-based optimization techniques, such as Hill Climbing and Simulated Annealing, directly into a model transformation approach.

In this work, we also consider the problem of finding the best orchestration of a given set of transformation rules as an optimization problem. Thereby, we aim at applying SBSE techniques for solving a reoccurring problem in the MDE domain, while at the same time we aim for a loose coupling between both worlds. In this sense, models and model transformations are defined in the model engineering technical space, and the orchestration of the rule applications is delegated to search-based optimization technologies. Depending on the desired goals for a particular scenario, our approach finds one or many solutions, i.e., rule application sequences in terms of rule orchestrations. Doing so allows us to reuse the same set of transformation rules for several scenarios, to explicitly define the transformation goals for each specific scenario, and to automatically find the best orchestration of rules. Furthermore, by combining SBSE with model transformations, the developer can stay in the model engineering technical space. This means that the problem, the search configuration input parameters, and the computed solutions are defined at the model level.

2.3 Motivating example

In this section, we introduce the running example for demonstrating MOMoT. We selected the example from the model quality assurance domain. It is well known that the quality of an object-oriented design has a direct impact on the quality of the code produced. The Class Responsibility Assignment (CRA) problem [8] deals with the creation of such high-quality object-oriented models. When solving the CRA problem, one has to decide where responsibilities, in the form of class methods and attributes they manipulate, belong and how objects should interact [51].

2.3.1 Modeling the CRA problem

As running example, we use a simplified version of the CRA problem. As given elements, we have a set of methods and attributes as well as dependencies between them. Such structure is also referred to as responsibilities dependency graph (RDG). Based on the RDG, the goal is to generate a high-quality class diagram (CD). For this purpose, a RDG2CD model transformation is needed to evolve a RDG into a CD. Figure 5 depicts the meta-models that are used to represent RDGs and CDs, respectively. The RDG meta-model is shown in the upper part containing only the features and their dependencies, while the additional concepts of CD are shown in the lower part. As both languages share a large portion of the modeling concepts—actually the RDG meta-model is a subgraph of the CD meta-model—we use the package merge

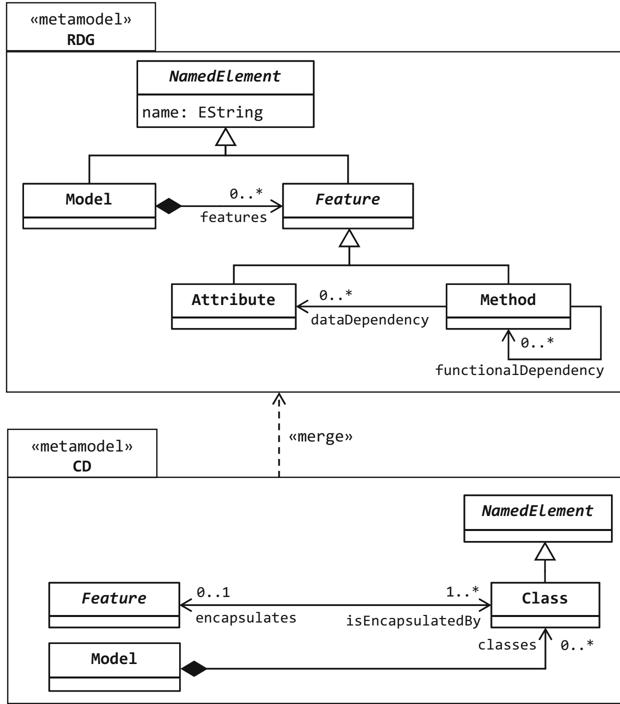

Fig. 5 RDG/CD meta-models

relationship to let the CD meta-model receive the elements of the RDG meta-model. In particular, the CD meta-model introduces the possibility that classes encapsulate features.

2.3.2 Transformation goals

The goal is to produce high-quality CDs from RDGs. The CRA problem is a problem with a fast growing search space of potential class partitions given by the Bell number $B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$. Already starting from a low number of features, the number of possible partitions is unsuitable for exhaustive search, e.g., 15 features yield 190899322 possible ways to create classes.

For determining the quality of the obtained CDs, we use two common metrics for assessing the quality of grouping functionality into classes: coupling and cohesion [8]. *Coupling* refers to the number of external dependencies a specific group has, whereas *cohesion* refers to the dependencies within one group. Typically, low coupling is preferred as this indicates that a group covers separate functionality aspects of a system. On the contrary, the cohesion within one group should be maximized to ensure that it does not contain parts that are not part of its functionality. Mapping these definitions to our problem, we can calculate coupling and cohesion as the sum of external and internal dependencies, respectively. The CRA problem uses the coupling and cohesion ratios, i.e., the coupling and cohesion achieved considering the total number of classes and attributes. All the formulae to calculate the cohesion ratio ($CohRt$) and coupling ratio ($CoupRt$) are shown in Fig. 6 (taken from [51]).[3] Please note that $M(c)$ and $A(c)$ refer to all methods and attributes of class c, respectively, MMI(c_i, c_j) and MAI(c_i, c_j) indicate the number of method–method and method–attribute interactions between classes c_i and c_j, respectively, and m_i and a_j refer to the ith method and jth attribute, respectively.
Summing up, the challenge of this case is to find a way to properly orchestrate transformation rules to optimize the quality of the produced CDs.

3 MOMoT

In this section, we first present a glance at MOMoT, by describing what it is and what it is made for. Then, we explain MOMoT's architecture, its current state and mention the extensions performed in this work. Finally, we describe a tour on MOMoT by giving some insights on its functioning. We exemplify some parts with our running example, namely the CRA problem.

3.1 MOMoT at a glance

With MOMoT, we have developed a novel approach for search-based model transformations, which builds on the non-intrusive integration of search-based optimization and model transformations to solve complex problems on model level. In particular, we formulate the transformation orchestration problem as a search problem by providing a generic solution encoding based on transformations, which allows for search-based exploration of the transformation space and explicating the transformation objectives. The idea to tackle this problem using search-based techniques is also reflected in the recent approaches by Abdeen et al. [1] who integrate multi-objective optimization techniques to drive a rule-based design exploration. Denil et al. [17] also address this problem by integrating single-state search-based optimization techniques, such as Hill Climbing and Simulated Annealing, directly into a model transformation approach.

Based on this trend and the ideas that we have initially outlined in [45] on combining meta-heuristic optimization and MDE, similar to Search-Based Software Engineering (SBSE) [35], we introduced a problem- and algorithm-agnostic approach called *MOMoT* (Marrying Optimization and Model Transformations) [25,28]. Our approach loosely couples the MDE and search-based optimization worlds to allow model engineers to benefit from search-based techniques while staying in the model engineering technical space, i.e., the input, the search configuration, and the com-

[3] Zero is assigned to the result of a division whenever its denominator is zero.

$$CohRt = \sum_{c_i \in Classes} \frac{MAI(c_i, c_i)}{|M(c_i)| \times |A(c_i)|} + \frac{MMI(c_i, c_i)}{|M(c_i)| \times |M(c_i) - 1|}$$

$$CoupRt = \sum_{\substack{c_i, c_j \in Classes \\ c_i \neq c_j}} \frac{MAI(c_i, c_j)}{|M(c_i)| \times |A(c_j)|} + \frac{MMI(c_i, c_j)}{|M(c_i)| \times |M(c_j) - 1|}$$

$$MMI(c_i, c_j) = \sum_{\substack{m_i \in M(c_i) \\ m_j \in M(c_j)}} DMM(m_i, m_j)$$

$$MAI(c_i, c_j) = \sum_{\substack{m_i \in M(c_i) \\ a_j \in A(c_j)}} DMA(m_i, a_j)$$

$$DMA(m_i, a_j) = \begin{cases} 1 & \text{if there is a dependency between } m_i \text{ and } a_j \\ 0 & \text{otherwise} \end{cases}$$

$$DMM(m_i, m_j) = \begin{cases} 1 & \text{if there is a dependency between } m_i \text{ and } m_j \\ 0 & \text{otherwise} \end{cases}$$

Fig. 6 Formulae for calculating the cohesion ratio ($CohRt$) and coupling ratio ($CoupRt$)

puted solutions are provided at model level. In particular, we are focusing on how different meta-heuristic methods can be used to solve an optimization problem, on how we can support the model engineer in configuring these methods, and on the separation of the objectives of the transformation from the transformation itself. This separation enables the reuse of the same set of transformation rules and objective specifications for several problem scenarios.

MOMoT offers the following features for developing search-based model transformations: (i) a generic way to describe the problem domain and the concrete problem instance, (ii) an encoding for the solution of the concrete problem instance based on model transformation solutions, (iii) a random solution generator that is used for the generation of an initial, random individual or random population, and (iv) a set of search-based algorithms to execute the search. To further support the use of multi-objective evolutionary algorithms, we additionally provide (v) generic objectives and constraints for our solution encoding, (vi) generic mutation operators that can modify the respective solutions, and (vii) a configuration language that also provides feedback about the specified search configuration.

Since our approach combines MDE techniques with SBSE techniques, the key building blocks are an environment to enable the creation of meta-models and models, a model transformation engine and language to manipulate those models and a set of meta-heuristic algorithms that perform a search to find transformation orchestrations that optimize the given objectives and fulfill the specified constraints.

3.2 MOMoT's architecture

MOMoT's technology stack is depicted in Fig. 7. In order to unify the MDE and SBSE worlds in a single framework, we bridge the Eclipse Modeling Framework (EMF), the Henshin graph transformation framework, and the MOEA meta-heuristic search framework.

The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. At the core of EMF is Ecore, which enables the definition of meta-models. Ecore is the de-facto reference implementation of the Essential Meta Object Facility standard in Java. By basing our implementation of MOMoT on EMF, we can use a multitude of existing frameworks.

Henshin [4] offers a rich language and associated tool set for in-place transformations of Ecore-based models. Henshin comes along with a powerful declarative model

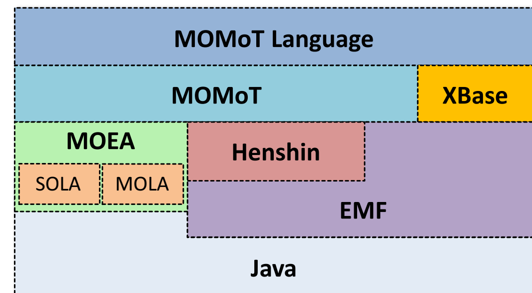

Fig. 7 MOMoT's architecture

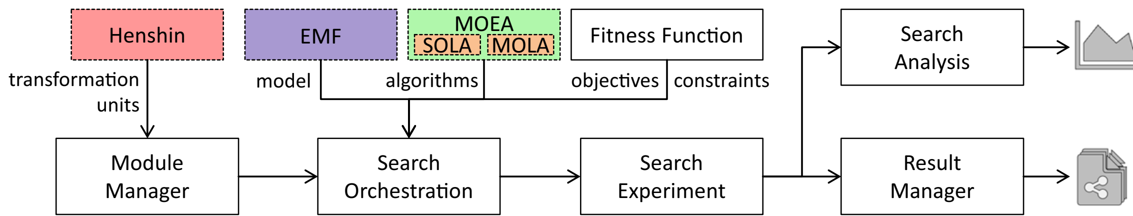

Fig. 8 Overview of MOMoT's workflow

transformation language that has its roots in attributed graph transformations and offers the possibility for formal reasoning. Besides rules, Henshin provides units to orchestrate these rules, e.g., sequential units, priority units or amalgamation units [7].

MOEA is the base meta-heuristic framework used in MOMoT. It is an open-source Java library that provides a set of multi-objective evolutionary algorithms with additional analytical performance measures and can be easily extended with new algorithms. Several different search algorithms can be used for investigating the search space, including global evolutionary search algorithms that are offered out-of-the-box by MOEA, such as NSGA-III and ε-MOEA. For the non-problem-specific search operators used by these evolutionary algorithms, i.e., the selection operator and the recombination operators, we reuse the generic operators provided by MOEA, such as the tournament selection operator and the one-point crossover recombination operator. For the problem-specific mutation operator, we have implemented three dedicated mutation operators that take the semantics of transformation units into account. The first operator replaces random transformation units by placeholders reducing the actual solution length. The second operator varies the user parameters of a transformation unit based on the parameters' values. And the third operator selects a random position within a solution and replaces all transformation units after that position with a random, executable transformation unit. The specific selection, recombination and mutation operations to use for a particular model transformation problem have to be specified in the search configuration described in the end of this section. Please note that additional selection, recombination and mutation operators can be easily plugged into MOMoT by implementing the Java interface *Variation*[4] provided by MOEA for this purpose.

In our previous work [25], we also included in MOMoT *single-objective local-search algorithms* (SOLA) for solving problems with only one objective, namely Hill Climbing and Random Descent. As a novel feature of MOMoT, in the present work we have implemented *multi-objective local-search algorithms* (MOLA) for model transformations. Indeed, with the aim of trying to discover the best algorithm for each particular scenario, we aim at studying how effective local-search algorithms can be for solving problems with more than one objective, and checking if they can be more effective than multi-objective algorithms for solving specific problems. For this reason, in this paper we contribute different multi-objective local-search approaches, which have not been supported by MOMoT before, with the aim of improving MOMoT's capability to find good model transformation orchestrations. The extensions to MOMoT are explained in detail in Sect. 4.

For specifying search-based model transformations, MOMoT provides the *Search Configuration Modeling Language* (SCML), which is exemplified in Sect. 3.3. SCML is built upon the functionality of XBase and provides a model-based representation of search configurations. Therewith, it provides dedicated support for transformation engineers to make use of search-based algorithms. Such search configurations include, for instance, the specification of the search algorithm to use and configurations needed for the respective search algorithm, such as the aforementioned search operations used by evolutionary algorithms or the population size used by population-based algorithms.

The complete source code of MOMoT with further explanations, as well as the cases currently realized with MOMoT, can be found on our project Web site.[5]

3.3 A tour on MOMoT

In this section, we give some details of how MOMoT works, exemplifying some parts with the CRA example. Figure 8 shows the typical MOMoT workflow as well as the involved artifacts. Some of its parts are described in the following.

3.3.1 Problem encoding

As it is typical in MDE, the problem domain itself is defined as a meta-model (cf. the meta-model of the CRA problem depicted in Fig. 5). Based on the specific problem domain, a user can define both concrete problem instances, i.e., models, and transformation rules that specify how problem instances

[4] http://moeaframework.org/javadoc/org/moeaframework/core/Variation.html.

[5] http://martin-fleck.github.io/momot/.

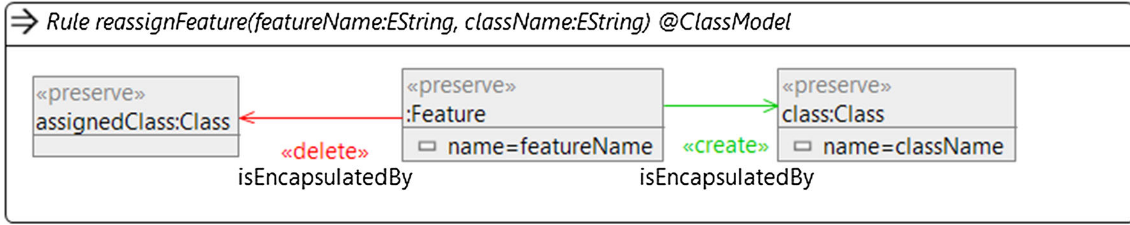

Fig. 9 Implementation of the reassign rule in Henshin

can be modified in order to produce a solution, i.e., an output model. In order to develop the necessary transformation rules, MOMoT reuses Henshin. Furthermore, since in our approach we separate the objectives from the rules, no further adaptations to those rules are necessary. The rule required for the CRA example is depicted in Fig. 9.[6] In the rule, features with a gray background represent elements that are in the model and remain in the model after a rule application, those in red represent elements that are in the model before the rule application and that are deleted afterward, and green elements were not in the model before the rule application but appear after the rule application. Therefore, the rule depicted in Fig. 9 models the reassignment of the feature received as parameter, *featureName*, to the class received as parameter, *className*, no matter to which class the feature was assigned before (this is the reason why we do not need the class on the left as parameter in the rule). As we start with a random CRA solution, which is improved by running the transformation, we simply need one rule that reassigns the features between different classes.

3.3.2 Solution representation

A solution in general consists of a set of decision variables that are optimized by the respective SBSE algorithm, a number of constraints that need to be fulfilled in order for the solution to be valid, and a number of objective values, one for each of the objective dimensions evaluated by the defined fitness function. As we deal with a transformation problem, there are two common ways to represent a solution. Either a solution is an ordered sequence of rule applications or it is the model resulting from the application of that sequence. We chose the first encoding as we consider it more flexible, because the resulting model can always be calculated from the sequence of configured rules and may be stored in a solution as attribute to avoid re-execution.

Furthermore, in our understanding, using a rule application sequence as first-class citizen in the encoding has several advantages. First, we are in line with the general SBSE problem formulation, where a solution consists of separate decision variables that are optimized by an algorithm. This increases the understanding for users who are knowledgeable in SBSE. Second, we are on the level of abstraction on which the user has provided information, i.e., transformation rules. Therefore, giving also novices in SBSE a bit of insight into the solutions. And third, we think that having the rule sequence shown explicitly also makes the output models more comprehensible as the user can compare solutions on the level they are computed and not only based on the output model, which may increase the acceptance of our approach.

Therefore, a decision variable in our solution is one transformation unit. MOMoT supports many different transformation units [28]: transformation rule, sequential unit, priority unit, independent unit, loop unit, iterated unit, conditional unit and placeholder unit.

3.3.3 Transformation unit parameters

Parameters allow to change the behavior of transformation units with variable information that is typically not present before execution time. When dealing with model transformations, we can distinguish between two kinds of parameters: those that are matched by the graph transformation engine (*matched parameters*), and those that need to be set by the user (*user parameters*). The former are often nodes within the graph, whereas the latter are typically values of newly created or modified properties. In the rule provided for the CRA problem (cf. Fig. 9), the *name* of the *:Feature* and the *name* of the *class:Class* are matched parameters, whereas the *featureName* and *className* parameters are user parameters.[7]

3.3.4 Solution repair

Even though constraints can be used to specify the validity or feasibility of solutions, a solution that is the product of recombining two other solutions (such as needed in evo-

[6] Please note that MOMoT supports different Henshin transformation units and more complex transformations. We refer the interested reader to [28].

[7] Please note that the *x:y* notation used in Henshin is inspired from the UML object diagram notation. Thus, it contains as first part the variable name used for the element match and as second part the type the element match has to conform to.

lutionary algorithms) might have unit applications that are no longer executable, e.g., because an object passed as a user parameter to the rule does not exist in the model anymore. By default, units that cannot be executed are ignored. However, this behavior might not be satisfactory in some cases as the process of determining that a transformation unit cannot be executed is quite expensive due to unnecessary match finding done by the transformation engine. Therefore, we consider two repair strategies in our approach.

The first strategy replaces all non-executable transformation units with transformation placeholders and the second strategy replaces each non-executable transformation unit with a random, executable transformation unit. In the ideal case, no solution repair strategy is necessary as non-executable unit applications are not produced. Of course, this depends on the chosen algorithm and the actual constraints of the solutions. A user can also select a dedicated recombination operator that is able to consider some constraints, e.g., the partially matched crossover (PMX) [34] can preserve the order of variables within a solution.

3.3.5 Search configuration

For configuring the search that should be used to find a good model transformation orchestration, we provide an own *Search Configuration Modeling Language* (SCML). As mentioned before, SCML is defined based on XBase and, thus, integrates Java into the language. Hence, Java can be used for defining different parts of the search configuration. Furthermore, we have also integrated OCL into SCML for allowing some parts of the configuration to be defined with OCL, especially the objectives and constraints defining when a found solution is considered as being "good".

The search configuration comprises the following parts: the fitness function to be applied in the search comprising objectives and constraints, the configuration of the search algorithm that should be used, the transformation input, the way in which the search results should be provided with possibly necessary post-processing steps, and the result analyses that should be carried out. These parts are explained in detail in the following paragraphs.

3.3.6 Objectives and constraints

The quality of each solution candidate is defined by a fitness function that evaluates multiple objectives and constraint dimensions. Each objective dimension refers to a specific value that should be either minimized or maximized for a solution to be considered "better" than another solution. For instance, if we have two solutions with a similar coupling value for the CRA problem, the one with the highest cohesion will be considered better. Additionally, a solution candidate may be subject to a number of constraints in order for the solution to be valid. Such constraints are not strictly enforced in the search, i.e., intermediate solution models as well as final solution models may violate such constraints, but they do impact the fitness of the solution. This is also true for constraints imposed by the meta-model defining the problem domain.[8] Depending on the algorithm, such invalid solutions may be filtered out completely or may receive a low ranking in relation to the magnitude of the constraint violation. For instance, we may decide to discard solutions where more than two classes are empty, or solutions where all features are within the same class.

As explained before, objectives and constraints can be defined either by providing Java implementations or by specifying model queries in OCL (the interested reader is referred to [28] for additional examples). The calculation of the objective values described in Fig. 6 as mathematical formulae has been implemented in Java for computing the coupling ratio and the cohesion ratio (cf. Listing 1; lines 3 and 4 refer to the Java class *FitnessCalculator* that computes these metrics). In lines 7–9, we show a constraint defining that the number of features not encapsulated by any class should be minimized. The computation of the not encapsulated features is defined with OCL (line 9).

We also provide default objectives such as done for the solution length (line 5), i.e., the length of rule application sequences of the computed solutions, which we aim at minimizing. There are several reasons for preferring shorter solutions, i.e., solutions consisting in the minimal possible rule applications. First, genetic algorithms may introduce dead genes, i.e., transformation rule applications that do not have any effect on the output. Second, shorter solutions may contribute to lowering the costs in cases where the model represents a physical system that has to be changed. Finally, shorter solutions are more easily understandable as the resulting model is closer to the initial version. Indeed, if models are changed as a result of an optimization process, these changes might need to be inspected by a person. Needing to understand less transformation steps and the models obtained after each step imply less efforts.

We also want to highlight that a specified fitness function may of course define conflicting objectives. This is why the provisioning of multi-objective search algorithms is an important feature of MOMoT. Such algorithms allow to define conflicting objectives separate from each other and perform the search in such a way that a good trade-off between them can be found. In particular, multi-objective search algorithms look for Pareto-optimal solutions, which are solutions where at least one objective cannot be further improved without worsening other objectives.

[8] The only exception is the distinction between single-valued and multi-valued features, which is strictly enforced by Henshin.

Listing 1: Specifying the Search Objectives

```
1  fitness = {
2   objectives = {
3    CouplingRatio : minimize { FitnessCalculator
       ↪.calculateCoupling(root) }
4    CohesionRatio : maximize { FitnessCalculator
       ↪.calculateCohesion(root) }
5    SolutionLength : minimize new
       ↪TransformationLengthDimension
6   }
7   constraints = { ...
8    UnassignedFeatures : minimize {
9     Feature.allInstances()->select(
       ↪isEncapsulatedBy == null)->size()
10  }
11 }
```

3.3.7 Search algorithm configuration

The search algorithm configuration defines the search algorithms to be used as well as their configurations. An example of such a configuration for our CRA example is the use of three algorithms that are executed sequentially (cf. Listing 2). In the example shown in the listing, we use NSGA-III and ε-MOEA for multi-objective search (lines 3 and 4, respectively), which is needed as we have three partially conflicting objectives (cf. Listing 1). In addition, we use random search as a baseline comparison to demonstrate the need for a meta-heuristic search (line 2). As we are using population-based algorithms, we have to configure the population size for each generation (set to 100, line 6) as well as the stopping criteria as maximum evaluations per run (set to 10,000, line 7). As meta-heuristic search includes some randomness, one may also define that the algorithms are executed several times to allow to draw statistical conclusions about the performance of the different algorithms. In this code excerpt, we establish 30 runs (line 8).

Listing 2: Configuring the Search Algorithms and Parameters

```
1 algorithms = {
2  Random:moea.createRandomSearch()
3  NSGAIII:moea.createNSGAIII()
4  eMOEA:moea.createEpsilonMOEA()}
5 experiment = {
6  populationSize = 100
7  maxEvaluations = 10000
8  nrRuns = 30 }
```

3.3.8 Transformation input

The execution of MOMoT transformations are started with dedicated run configurations that execute the compiled MOMoT search configurations, as shown in Listing 3. Please note that input models are modeled in EMF and encoded in XMI. In the listing, the input model, named *Initial_Config.xmi*, is loaded in line 2. In order to allow for an efficient search, a preprocessing is possible to prepare an initial structure beneficial to perform the search or to slice the model to reduce the memory consumption during the search process. In our example listing, line 4 specifies a loop for randomly adding some classes to the model (recall that, as explained in Sect. 2.3 and exemplified in Fig. 5, there should be no classes in the input model). Then, line 5 specifies another loop for distributing methods and attributes randomly among the created classes. Variable *cm* stores the root of the model, which is an element of type *Model* (cf. Fig. 5).

Listing 3: Defining the Transformation Input and Preprocessing

```
1 model = {
2  file = "problem/Initial_Config.xmi"
3  adapt = { var cm = root as Model
4   for(i:0 ..< cm.features.size - cm.classes.
       ↪size) ... // add randomly some classes
5   for(feature : cm.features) ... // distribute
       ↪ features randomly
6   return cm } }
```

3.3.9 Transformation results

MOMoT provides as transformation results: (i) the set of orchestrated transformation sequences leading to (ii) the set of Pareto-optimal output models with (iii) their respective objective values. The objective values may give an overview of how well the objectives are optimized. Listing 4 provides an excerpt of this configuration, and, in addition, shows how results may be post-processed and how specific solutions are selected.

In particular, line 3 shows a post-processing operation in which empty classes are removed from the model. Line 4 specifies where to store the objective values of the resulting models, which is stored as plain text. Then, lines 5 and 6 determine the folder to store the sequences of rule applications that lead to the optimal models and the optimal models obtained with such rules applications, respectively. Finally, lines 7–9 show a further post-processing operation with the aim of inspecting the kneepoint solutions. A kneepoint is a solution that has better fitness (w.r.t. a specific dimension of the solution space) than the neighborhood solutions. The number of nearest neighbors for which the kneepoint solution must be better is specified by the neighborhood size (line 8).

Listing 4: Defining the Transformation Output and Postprocessing

```
1 results = {
2  adaptModels = { //remove empty classes
3   root.classes.removeAll(cm.classes.filter[c |
       ↪ c.encapsulates.size == 0])}
4  objectives = { outputFile = "output/
       ↪objectives/objective_values.txt"}
5  solutions  = { outputDirectory = "output/
       ↪solutions/" }
6  models = { outputDirectory = "output/models/"
       ↪ }
7  models = { //select kneepoint models for
       ↪further inspection
8   neighborhoodSize = maxNeighborhoodSize
9   outputDirectory  = "output/models/kneepoints
       ↪/"}}
```

3.3.10 Results analysis

MOMoT performs additional analysis to give more insights into the computed solutions and the relative algorithm performance (cf. Listing 5). For instance, we can statistically analyze dedicated performance indicators, such as Hypervolume (see line 2), to compare the performance of different algorithms. This data can also be used to plot graphs to give a better overview about the analysis.

Listing 5: Defining the Statistical Analysis Methods

```
1 analysis = {
2   indicators = [ hypervolume
      ↪invertedGenerationalDistance ]
3   significance = 0.01
4   show = [ aggregateValues
      ↪statisticalSignificance individualValues
      ↪ ] ...}
```

We use three algorithms for the given example as can be seen in Listing 2, ε-MOEA, NSGA-III and Random Search (RS), and execute each algorithm 30 times. The results of the analysis are depicted in Fig. 10. We can clearly see that for the Hypervolume indicator, random search has the lowest and therefore worst value, while ε-MOEA has the highest value. A similar result is produced for the Inverted Generational Distance, where lower values are considered better. The fact that a meta-heuristic search outperforms random search is a good indicator that the problem is suitable for SBSE techniques. In order to investigate the results further, MOMoT provides several other features to test and compare different algorithms [25]. More details about evaluation possibilities are provided in Sect. 5, which is about an in-depth comparison of local- and global-search algorithms supported in MOMoT for three case studies.

4 Improving MOMoT's local search capabilities

As explained in Sect. 3.1, MOMoT counts on local-search algorithms for solving problems with one objective (SOLA) and global-search algorithms for problems with more than one objective using the MOEA framework. However, with the aim of trying to discover the best algorithm for each particular scenario, we aim at studying how effective local-search algorithms can be for solving problems with more than one objective, and checking if they can be more effective than multi-objective algorithms for solving specific problems. They conform what we call *Multi-Objective Local-search Algorithms* (MOLA, cf. Fig. 7).

Indeed, the existence of both local-search and global-search approaches raises the question of which approach is better for a specific application domain. In this work, we contribute with an empirical comparison in the domain of model transformations, for which we compare a set of local-search

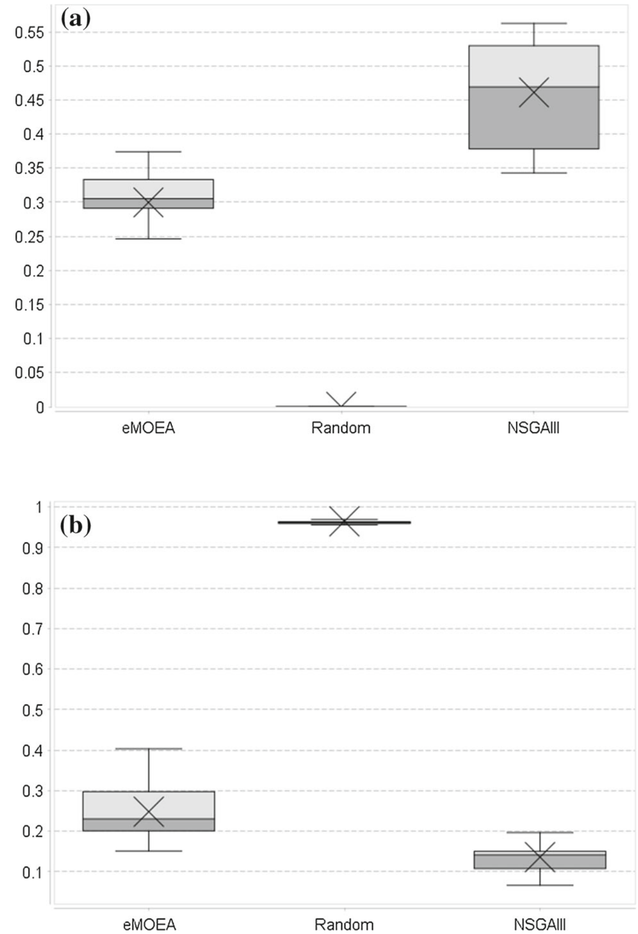

Fig. 10 Statistical analysis for the CRA case results. **a** Hypervolume indicator, **b** Inverted Generational Distance

approaches and a set of global-search approaches for three case studies. In order to implement MOLA in MOMoT, we need to go beyond local-search algorithms used to solve problems with one objective. For obtaining the Pareto-optimal set of solutions, we need to make these algorithms iterative, so that they do not focus on improving only one objective, but all of them. Besides, in order to obtain the best solutions, we need these algorithms not to stop in local optima.

In this section, we first explain how iterated local search has been implemented as an extension to MOMoT. Then, we describe further extensions to local-search algorithms necessary for managing to escape local optima. We conclude this section with a synopsis summarizing the improvements made to MOMoT regarding local-search capabilities.

4.1 Iterated local search

The idea is to build single objectives out of a set of objectives. This can be done by applying weights to the different objectives and aggregating them all into one objective to be

Algorithm 3: Iterated local search in MOMoT

```
   Output: Pareto-optimal set p
1  p = ParetoSet{}
2  repeat
3      s_0 = randomSolution()
4      (f, Δ) = randomAggregatedFitness()
5      b = localsearch(s_0, f, Δ, e_s)
           // Perform a local search with
              specific fitness function and
              evaluation count
6      e = e + localsearchevaluations
7      p = p ∪ {b}
8  until Evaluations e exhausted
```

either maximized or minimized. We have extended MOMoT by implementing such a mechanism.

A high-level description of the implementation of the iterated local search, the basis for MOLA, is shown in Algorithm 3. Basically, multi-objective solutions are found by iteratively applying single-objective local searches. As usual, we start with an empty set of solutions (line 1) and repeat the same process until the stopping criterion is met (lines 2–8). In this process, a random solution is obtained and is stored in s_0 (line 3). In every iteration, a different multi-to-single-objective aggregation is used as fitness function (variable f; line 4). In each of these aggregations, different random weights are given to the different objectives, with the purpose of obtaining the optimal solutions in the Pareto front and avoiding the optimization of only one objective. Since, iteration after iteration, we get an array of solutions (p), we use Δ to represent the improvement among solutions. In particular, having two solutions A and B, Δ specifies the greatest relative worsening of any objective of A with respect to B, meaning that A is not Pareto-dominating B. This Δ, therefore, determines the solutions to be gathered in p. With the explained inputs, an algorithm for local search is run (line 5, where e_s is used to denote the evaluation count, i.e., the number of fitness evaluations performed in this run) and a solution is obtained, stored in b and added to the Pareto set of solutions p (line 7).

4.2 Escaping local optima

In line 5 of Algorithm 3, we see that we apply a local-search algorithm. With the aim of avoiding to fall into and get stuck in local optima, we have developed another extension to MOMoT that consists in a tailored implementation of Simulated Annealing (SA) [46] as a new local-search algorithm. SA belongs to a class of local-search algorithms that are known as threshold algorithms [67]. It has a stochastic component. This algorithm was originally inspired from the process of annealing in metal work. Annealing involves heating and cooling a material to alter its physical properties due

Algorithm 4: Simulated Annealing in MOMoT

```
    Input: Initial Solution s_0
    Output: Best Solution b Found
1   t = 0, b = s_0, c = s_0, n_b = 0, n_c = 0
2   repeat
3       n = SelectAny(N(c))          // Select random
        neighbor
4       If f(n) > f(b) Then
5         b = c = n; n_b = n_c = 0
6       Elseif f(n) > f(c) Then
7          c = n
8          n_c = 0
9       Else
10         n_c = n_c + 1
11         If rnd() < e^(−Δf(n,c)·(t_s+t·t_d)) Then
12            c = n
13         Endif
14      Endif
15      If n_b > c_b or n_c > c_c Then
16         c = b
17         n_b = n_c = 0
18      Endif
19      t = t + 1
20      n_b = n_b + 1
21  until Evaluations exhausted
```

to the changes in its internal structure. As the metal cools, its new structure becomes fixed, consequently causing the metal to retain its newly obtained properties. In Simulated Annealing, we keep a temperature variable to simulate this heating process. It is initially set high and then it is allowed to slowly cool as the algorithm runs. While this temperature variable is high, the algorithm will be allowed, with more frequency, to accept solutions that are worse than our current solution. This gives the algorithm the ability to jump out of any local optima it finds early in its execution. As the temperature and therewith the chance of accepting worse solutions are gradually reduced, the algorithm gradually focuses on an area of the search space in which, hopefully, a global optimum can be found. This gradual cooling process is what makes the SA algorithm effective at finding a close-to-optimum solution when dealing with large problems that contain numerous local optima.

A pseudo-code representation of our tailored implementation of the SA algorithm in MOMoT is depicted in Algorithm 4. In this excerpt, t represents the time, which is initialized with 0 (line 1) and discretely increases in each iteration (line 19). Variable b keeps the best solution found and is initialized with the random solution s_0 (line 1). Variable c denotes the current solution, also initialized with s_0 (line 1). Finally, variables n_b and n_c denote the number of steps (iterations) performed since the latest best solution and current solution were set, respectively.

After these initializations (line 1), the following process is repeated until the stopping criterion is met (lines 2–21). An iteration starts by selecting and keeping in n a random

neighbor of the current solution. If the fitness of this neighbor is better than the fitness of the best solution so far (line 4), then we update the best and current solutions with such neighbor and set the number of iterations since the latest best and current solutions to 0 (line 5). If the fitness of the selected neighbor, n, is not better than the fitness of the best solution so far, but it is better than the current solution (line 6), then the current solution and the number of iterations since the current solution are updated accordingly (lines 7 and 8). If the fitness of the selected neighbor, n, is not better than the fitness of the best solution so far neither of the current solution (line 9), then the number of iterations since the current solution is increased (line 10).

Line 11 gives the stochastic behavior to our algorithm to decide whether to jump to a worse solution (if the algorithm reaches this line, it is because $f(n) < f(c) < f(b)$). Here, e is the Euler number. Regarding its exponent, $\Delta_f(n, c)$ denotes the fitness decrease from the fitness of the selected neighbor to the fitness of the current solution, t_s is the inverse of the start temperature (which is set to a high value), t is the current iteration, and t_d is a temperature decrease value. The two last variables model the decrease in the temperature over time. As the iterations increase, the second part of the multiplication increases. And, since this is part of an exponent that is transformed to a negative number, the higher the value of the exponent, the lower the final value of the computation and therefore the less chances to jump to the worse solution (line 11). This is how we emulate the annealing behavior. The fitness decrease also affects this decision, since it is also part of the exponent. Thus, the higher the fitness decrease, the less chances to jump to a worse solution.

As earlier mentioned, the comparison in line 11, $rnd() < e^{-\Delta_f(n,c)\cdot(t_s+t\cdot t_d)}$, is used to stochastically decide whether to select a different place in the solution space. The higher the exponent of the right-hand side of the comparison is, the lower is the value of the right-hand side (since it is applied the negative product) and, therefore, the lower the chances are to perform the jump. If the jump is to be done, then the current solution is set to the (worse) random neighbor (line 12) computed in this iteration. However, in this case please note that we do not reset n_c, since the fitness value of the current solution must have gotten worse.

In line 15, c_b and c_c are threshold values. They are used in lines 15–18 to decide whether the search restarts at the best solution found so far if no improvement to the current solution can be done within c_c iterations or no improvement to the best solution can be done within c_b steps. In lines 19 and 20, the discrete time and the number of iterations since the latest best solution are updated appropriately.

We have also extended the implementation of the Hill Climbing algorithm so that it aborts and thus restarts when no improvement can be found for n iterations. Furthermore, we have augmented both Simulated Annealing and Hill Climbing with a limited version of Tabu Search [32]. Tabu Search helps avoiding local optima by using different memories to remember the recently visited solutions (short term), focus or prohibit moves toward solutions with specific characteristics (intermediate). In particular, we forbid to change variables that have been recently changed by modifying the neighborhood of each search to look for rule additions, modifications or deletions for forbidden variables.

Finally, we have also added a new way of obtaining the neighborhood population for the local-search algorithms. The neighborhood population contains random solutions where parameters of a rule application have changed or the rule application itself has been replaced or deleted. There is a parameter *end* to favor (i) adding new transformations at the end, (ii) replacing an empty rule application before or (iii) removing an arbitrary one in case no such rule application exists. This has the advantage that no rule can get broken as a result of this rule application, therefore preserving as many existing transformations as possible. Consequently, the effect of adding this single rule application may behave better in certain scenarios.

4.3 Synopsis

So far, local-search algorithms were applied in MOMoT for solving problems with only one objective, what we know as *single-objective local-search algorithms* (SOLA, cf. Fig. 7). In particular, only Hill Climbing [11] and Random Descent were included in MOMoT for solving single-objective problems. But the problem with these two local-search algorithms is that they can fall in local optima quite fast without a chance to escape. In fact, after starting in an initial random solution, they always select a neighbor with a better fitness than the current solution. With this behavior, they may find a local optimum, from which they cannot continue (cf. Fig. 3). Furthermore, they were used to solve problems with only one objective.

In this extension of MOMoT, we have implemented iterated local search with the aim of exploring the usefulness of local-search algorithms for solving multi-objective problems, as described in Sect. 4.1. We have named this extension *multi-objective local-search algorithms* (MOLA, cf. Fig. 7). Furthermore, due to the limitation of current local-search algorithms to fall into local optima, we have implemented new versions of several of these algorithms with the aim of escaping local optima, as explained in Sect. 4.2. In particular, we have implemented new versions of Simulated Annealing and Hill Climbing, for which we have, among other techniques, integrated some features of Tabu Search. Please note that our iterated local search is easily extensible, since any local-search algorithm can be used. Finally, we have also implemented a new way of obtaining the neighborhood population, again with the aim of escaping local optima.

5 Case-study-based evaluation

In this section, we perform experiments based on the guidelines for conducting empirical explanatory case studies [58]. The main goal is to evaluate whether local- and global-search algorithms show different behaviors, i.e., in terms of search exploration and runtime performance, for orchestrating model transformation executions. Please note that this is the first study of its kind investigating and comparing the performance of single-objective local search, multi-objective local search, and global-search algorithms for their application to model transformations. This evaluation has been carried out based on three case studies considering three different transformations which are solved in a search-based manner with MOMoT: *Stack Load Balancing*, *Class Responsibility Assignment*, and *Transportation Line Optimization*. While the first two case studies have been already investigated in our earlier publications [25,28], they have not been used to compare the performance of local- and global-search algorithms as done in this study.

5.1 Research questions

As mentioned above, we performed this study to evaluate the possible distinct behavior of different search-based algorithms concerning their search capabilities for finding good solutions for transformation problems. More specifically, we aimed to answer the following two research questions (RQs):

- *RQ1—Search exploration* Is there a significant difference of local-search approaches to global-search approaches applied to the domain of model transformations concerning the exploration of the search space?
- *RQ2—Search time* Is there a significant difference of local-search approaches to global-search approaches applied to the domain of model transformations concerning the execution time of the search process?

In order to answer these research questions, we perform this study to extract several measurements using three case studies. The complete case study setup is summarized in the next subsection.

5.2 Case study setup

We now introduce the case studies used in our study, the measurements as well as statistical tests and finally the parameter configuration used for the search algorithms.

5.2.1 Case studies

Our research questions are evaluated using the following three case studies. Each one consists of one or more Henshin model transformations, the domain meta-models and a set of sample input models of different sizes. We have selected these cases due to their difference in used transformation concepts such as rule parameters, preconditions, and also number of objectives to consider.

- *CS1—Stack load balancing (SLB)* This transformation is about balancing the loads of different stacks by moving elements to the left or right neighbor stack. The description of the example is presented below.
- *CS2—Class responsibility assignment (CRA)* This transformation is the running example already presented in Sect. 2.3 and therefore not further introduced in this section.
- *CS3—Transportation line optimization (TLO)* This transformation is about finding the best configuration of a transportation line, considered within the domain of cyber-physical production systems, in order to speed-up the production process while minimizing the costs of producing items. It is described in detail below.

The stack load balancing case
This case study consists of a system of stacks, where each stack can have a different number of boxes referred to as *load*. The meta-model that represents the stack system is depicted in Fig. 11. Every stack in the system has a unique identifier, a number that indicates its load, and is connected to a left and right neighbor in a circular manner. A concrete stack example instance composed of five stacks with different loads is shown in Fig. 12 in its abstract syntax.

The objective is to minimize the load difference among the different stacks in the minimum number of steps, as specified in Listing 6. To manipulate stack models, we propose two basic rules which shift part of the load from one stack either to the left or to the right neighbor. The *ShiftLeft* rule is shown in Fig. 13—an analogous rule is used to shift parts to the right. Each rule has five input parameters: *fromId*, *toId*, *amount*, *fromLoad*, and *toLoad*. While producing a match for this rule, all these input parameters acquire a value, i.e., they are instantiated, and the rule can be applied. For retrieving these values, Henshin matches the pattern in the rule consisting of nodes and edges with the model graph. Since stacks are nodes in the graph and the left and right relationships are edges, they can be matched automatically and values for *fromId*, *toId*,

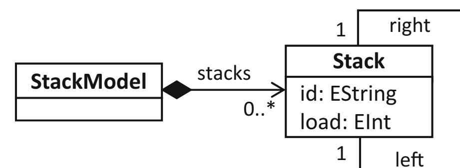

Fig. 11 Stack meta-model

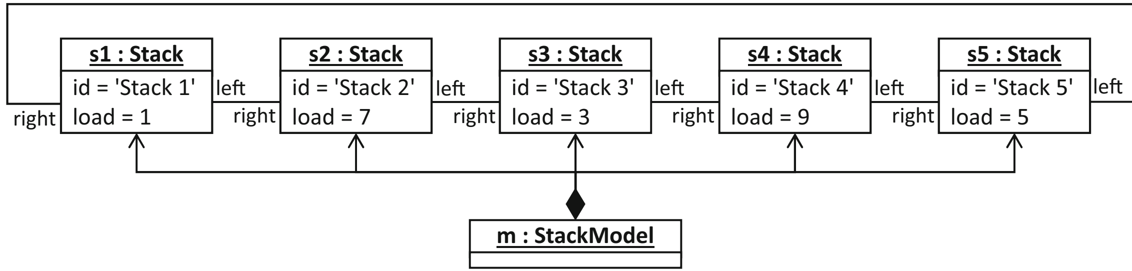

Fig. 12 Stack example model in abstract syntax

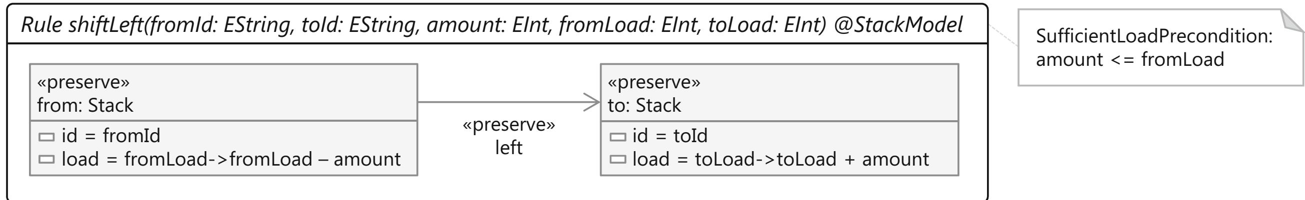

Fig. 13 *ShiftLeft* rule to shift load

Table 1 Input models of the stack case

Model	#Stacks	Load
A	2	11
B	5	28
C	10	50
D	25	139
Asc	10	45

Table 2 Input models of the CRA case

Model	#Methods	#Atts	#D. deps	#F. deps
A	2	3	4	0
B	4	5	8	7
C	7	9	19	16
D	15	18	36	53

fromLoad, and *toLoad* can be set. Furthermore, the rule also contains a precondition, which ensures that the *amount* that is shifted is not higher than the load of the source stack. This attribute condition is shown as an annotation in the figure.

We have used five different initial models of different sizes for our experiments in this case. They have from two stacks and a load of 5, to 25 stacks and a load of 139. Stacks *A* to *D* just contain a random number of loads, while *Asc* contains an ascending and descending number of loads 1-2-3-4-5-6-5-4-3-2 and was handcrafted to be unbalanced and at the same time to have no direct improving neighbors. Table 1 shows the characteristics of each model.

Listing 6: Stack Example - Search Objectives

```
1 fitness = {
2  objectives = {
3   Load : minimize { FitnessCalculator.
      ↪loadDifference(stackModel) }
4   SolutionLength : minimize new
      ↪TransformationLengthDimension
5  }
6 }
```

The class responsibility assignment case
This case has already been introduced as motivating example. For this case, we use four initial models of different sizes for our experiments. The characteristics of each model are shown in Table 2.

The transportation line optimization case
This case is taken from the research project CDL-MINT[9] carried out by TU Wien. The main goal is to investigate the application of MDE techniques in the domain of smart production. This case is about designing a cyber-physical production system in which different configurations for a production line are possible depending on the given objectives. Its meta-model is shown in Fig. 14. We distinguish three types of classes. While those with a gray background are abstract classes, the ones with a white background can be instantiated. Finally, classes in orange are used to represent non-functional properties. When instantiating a model, these classes get associated with an element (*System*, *Component* or *Item*) and represent several features of it at runtime.

In the meta-model, *System* is the root class, which is composed of several *Area*s. The system can have associated a *SimConfig*, where several parameters of the simulation can be specified, for instance, the simulation speed. The purpose of other parameters is to be calculated at runtime in order to measure some non-functional properties of the

[9] https://cdl-mint.big.tuwien.ac.at.

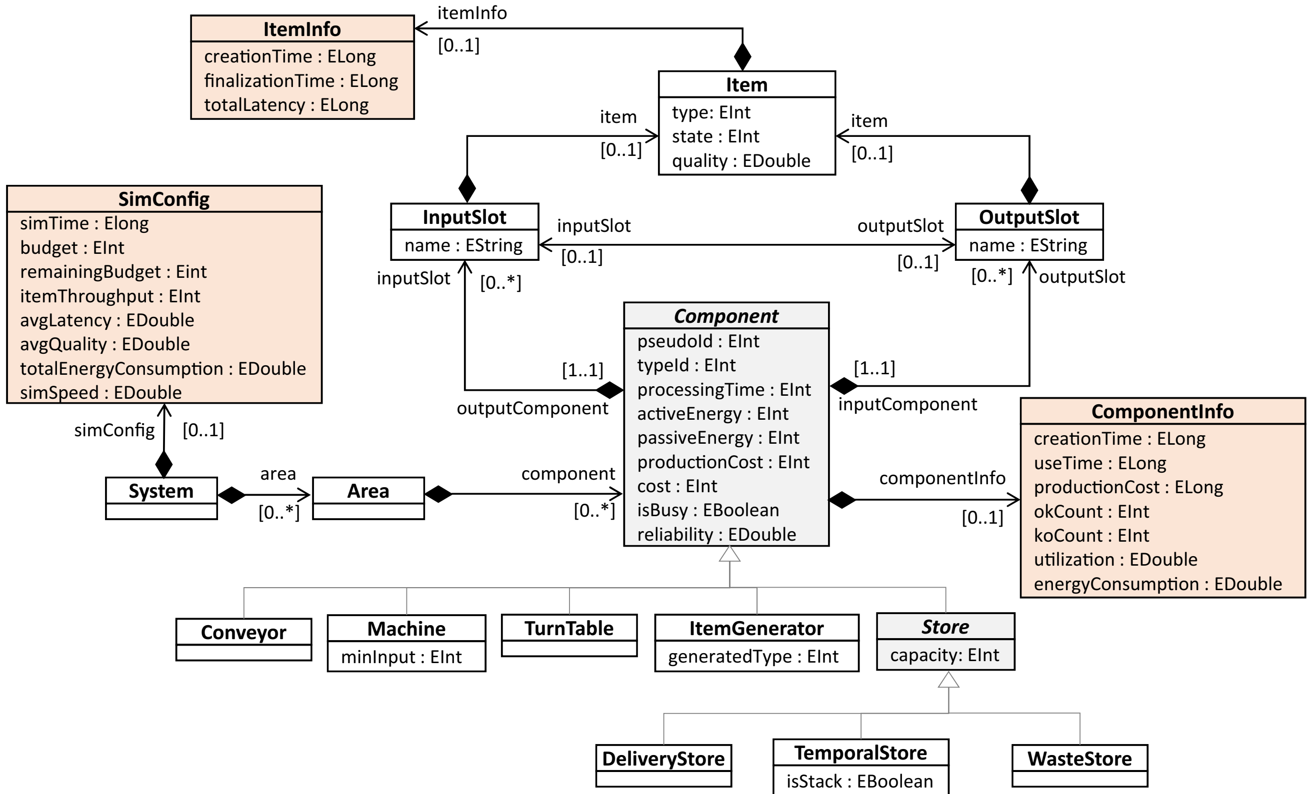

Fig. 14 Transportation line meta-model

system, such as the throughput or the latency. An area, in turn, can contain any number of *Components*. As we see, there are seven types of *Components*, namely *Conveyor*, *Machine*, *TurnTable*, *ItemGenerator*, *DeliveryStore*, *TemporalStore* and *WasteStore*. Each machine has its purpose. Conveyors, for instance, are used to move items along them, item generators are used to build items, and turn tables route items of a specific state or type to a dedicated location. Components can have associated a *ComponentInfo* entity, where several runtime parameters of the component are stored. This includes meta-information about objects like the creation time. Some other parameters are used to measure non-functional properties of the component during simulation time, such as its utilization or its energy consumption. Each component can have one or more *InputSlots* and *OutputSlots*. These are used to connect components among them. InputSlots and OutputSlots specify the required or respectively the provided type and state of the objects passed along. Items coming into an input slot of the wrong type are considered to be destroyed. The slots, in turn, can have an incoming or outgoing *Item*, and items can have an associated *ItemInfo* to store some non-functional information. *Items* have a *quality* attribute that, for the evaluation in this paper, only acquires values of 0 (low quality) and 1 (high quality). They are created by *ItemGenerators* and are moved along the transportation line. On their way, they may serve as input to *Machines*, which produce other items as output. Those items that complete the transportation line successfully, should end up in a *DeliveryStore*, representing the place where they are ready to be sent to the customer. If there is a problem with an item, this should end up in a *WasteStore* and must be discarded.

We use two rule sets for this meta-model that differ in the size of the search space. Four rules that are present in both rule sets are shown in Fig. 15. The rule in Fig. 15a, *createExpensivePrinter* is used to create an item generator that represents an expensive printer. In order for the rule to be executed, there must exist an area, so that the created printer gets associated with it. We also see in the figure how an output slot is associated with the added printer. Please note that no input slot is needed, since the printer will not receive any input. The rule contains a negative-application condition (NAC), represented by an element in blue. Thus, the rule will not be applied if there is a component, no matter of which specific type, whose *pseudoId* is the Integer received as parameter and whose *typeId* is 1. Please note that these values are precisely given to the created printer.

In the search of the optimal configuration, it may be necessary to remove existing components. For that purpose, we have the rule *deleteComponent* (cf. Fig. 15b). It deletes the

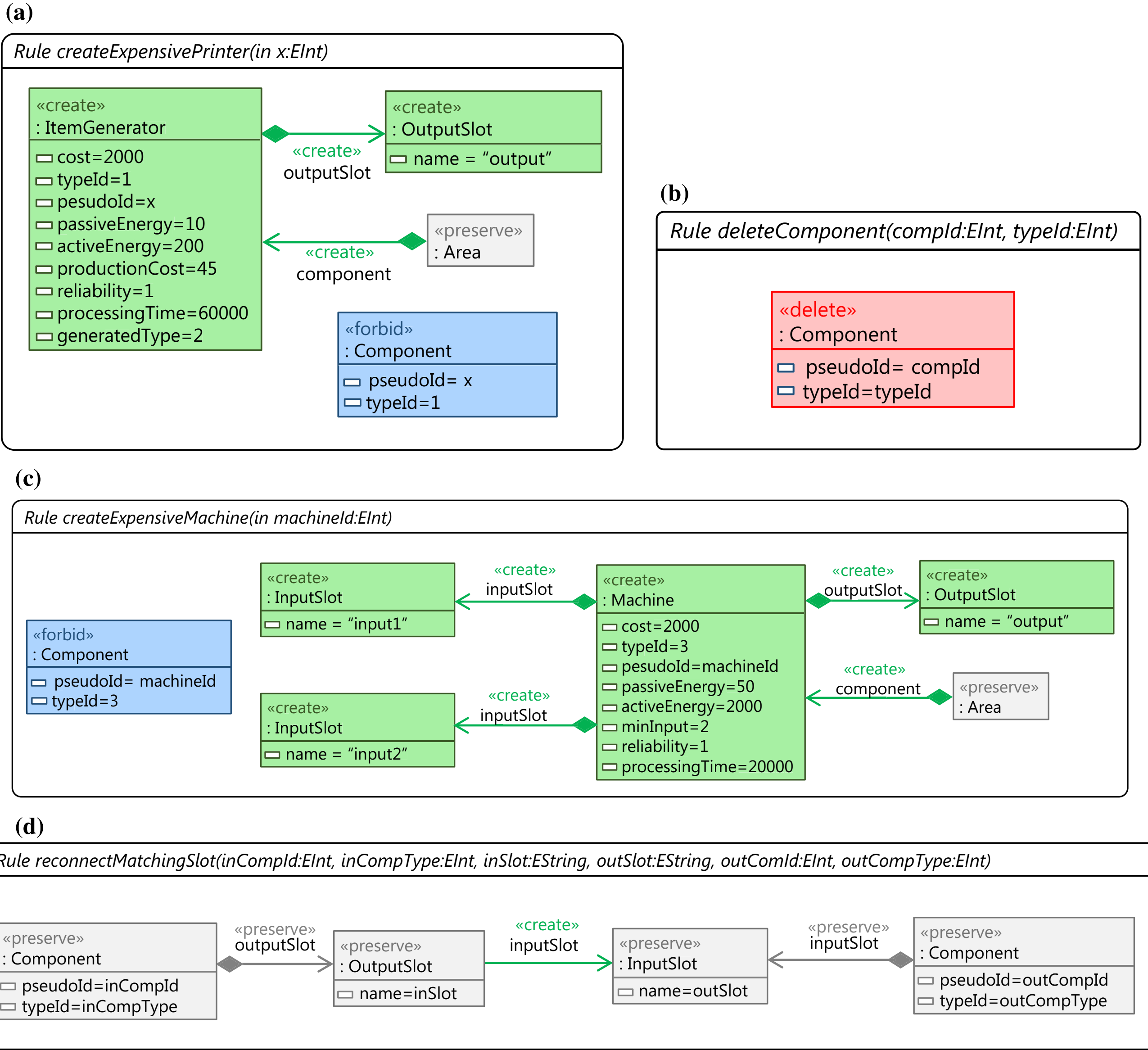

Fig. 15 Some rules of the transformation line example

component whose *pseudoId* and *typeId* match the parameters received. Similar to the rule for creating an expensive printer, we have rule *createExpensiveMachine* in Fig. 15c. Specifically, this is a type of machine that must have at least two inputs (attribute *minInput*) in order to produce output. Indeed, as we can see in the rule, two input slots are created by the rule and associated with the created machine. The NAC specifies that, in order to apply this rule, there must not exist in the system a machine with the *pseudoId* and *typeId* that are to be given to the new machine.

Finally, rule *reconnectMatchingSlot*, in Fig. 15d, is used to connect two components. In particular, it creates a connection between an output slot of a component and an input slot of another component. As we see in the rule, the parameters received by the rule are used to match the four specific entities.

For our evaluation, we use two different rule sets, namely a *generic* and a *specific* rule set, for reaching search spaces of different sizes. Both of them are used to solve two different problems, namely *creating* and *changing* a factory layout, resulting in four different settings.

Both rule sets contain the rules mentioned above, but differ in the remaining rules. In particular, the first, so-called *generic*, rule set can create arbitrary factories. It does so by first creating components with input and output slots and then connecting output slots with matching input slots. Conveyors and stores start with undefined slot types (− 1), while the other components have defined slot types. In the matching

process, such a component will get all its slot types assigned if the input component's slot type is defined.

The second, so-called *specific*, rule set only allows to create a limited set of factories to reduce the size of the search space. In particular, it only considers printers, machines and conveyors. The creation rules for printers and machines are equal to the creation rule of the generic case. However, there are two rules for conveyors transporting (a) parts and (b) complete toys. Thus, there are no slots with undefined slot types and there is only a single rule connecting matching slots.

As for the two problems to be solved, the first one is to *create* an optimal transportation line from scratch, while the second one is to *change* an existing transportation line, i.e., to find an optimal reconfiguration.

For the problem of creating a transportation line from scratch, we use a near-empty model as input. It only consists of a system with only one area, a waste store with two input slots and a delivery store with two input slots allowing only finished items. We do not use the rule deleting components. In the reconfiguration problem, we have applied a search process with the aim of producing an expensive transportation line. The outcome is the model that we use as input. It is composed of 42 components (13 item generators, 7 conveyors, 14 machines, 2 turn tables, 4 stacks, 1 waste store and 1 delivery store), 52 input slots, 45 output slots and only 7 slot connections. An optimization strategy will thus need to focus on finding the right connections and deleting unnecessary machines.

We have five objectives in this case, namely (i) maximizing the number of high-quality items in *DeliveryStore*s, (ii) minimizing the number of wasted items, (iii) minimizing the number of low-quality items in *DeliveryStore*s, (iv) minimizing the energy consumption of the system, and (v) minimizing the total cost of the items produced. The search objectives for this case have been written in Java. Many objectives contain additional code paths to avoid returning zero. For example, when checking for produced items, we consider the number of items that are in a *DeliveryStore*. However, if that number is zero, then the number of items that have been generated and are within the transportation line is used. We shall highlight that the calculation of the fitness is realized so that the first computation is preferred over the second one. Listing 7 shows the Java code for the first objective, namely maximizing the number of high-quality items in *DeliveryStore*s. Listing 8, in turns, shows a short OCL version of the objectives without any additional code to guide the search. Specifically, the objectives, in the order stated before are shown from line 4 to line 9.

Listing 7: Transportation Line Optimization Case - 1st Objective

```
1  fitnessFunction.addObjective(new
     ↪AbstractEGraphFitnessDimension("Items
     ↪produced") {
2  @Override
3  protected double internalEvaluate(
     ↪TransformationSolution solution) {
4    EGraph root = solution.execute();
5    System system = getSystem(root);
6    simulate(system);
7    int ret = getItemThroughput(getSimConfig(
     ↪system));
8    if (ret == 0) {
9      int unfinishedItems = 0;
10     int finishedItems = 0;
11     for (Area area: system.getArea()) {
12       for (Component comp: area.getComponent()
     ↪) {
13      ComponentInfo inf = comp.getComponentinfo
     ↪();
14      if (comp instanceof Machine && inf !=
     ↪null) {
15        finishedItems += inf.getOkCount();
16      }
17      if (comp instanceof ItemGenerator && inf
     ↪!= null) {
18          unfinishedItems += inf.getOkCount();
19      }
20        }
21     }
22     if (finishedItems != 0) {
23       return 10*(1+1.0/finishedItems);
24     } else if (unfinishedItems != 0) {
25       return 10*(2 + 1.0/unfinishedItems);
26     }
27     return 30;
28   }
29   return -ret;
30 }
31 });
```

5.2.2 Evaluation measures

To answer our research questions, we use several metrics depending on the nature of the research question.

In order to evaluate research question RQ1, we compare the results of NSGA-III, Iterated Simulated Annealing and Iterated Hill Climbing based on Hypervolume (IHV) and Inverted Generational Distance (IGD) for all cases. Hypervolume corresponds to the proportion of the objective space that is dominated by the Pareto front approximation returned by the algorithm and delimited by a reference point. The larger the proportion, the better the algorithm performs. It is interesting to note that this indicator is Pareto dominance compliant and can capture both the convergence and the diversity of the solutions. Therefore, IHV is a common indicator used when comparing different search-based algorithms. However, due to lack in solution variability, the hypervolume indicator is often zero. Thus, we additionally show the Inverted Generational Distance (IGD), i.e., the average distance of any solution vector in the approximated Pareto front to a vector in the solution space.

In order to evaluate research question RQ2, we compare the runtime of all algorithms. We measure the time between the start of the algorithm initialization and when we get the result. We do not consider the problem setup, i.e., neither the model loading nor the transformation loading, since it is equal for all algorithms. We do not consider the result analysis, including saving models and objective values, nor the warm up phase, either.

5.2.3 Statistical tests

Since meta-heuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our study is performed based on 30 independent simulation runs for each case and the obtained results are statistically analyzed by using the Mann–Whitney U test [3] with a 99% significance level ($\alpha = 0.01$). The Mann–Whitney U test [50], equivalent to the Wilcoxon rank-sum test, is a nonparametric test that allows two solution sets to be compared without making the assumption that values are normally distributed. Specifically, we test the null hypothesis (H_0) that two populations have the same median against the alternative hypothesis (H_1) that they have different medians. The p value of the Mann–Whitney U test corresponds to the probability of rejecting the H_0 while it is true (type I error). A p value that is less than or equal to α means that we accept H_1 and we reject H_0. However, a p value that is strictly greater than α means the opposite.

For each case, we apply the Mann–Whitney U test to compare the results of all algorithms. In particular, we want to see whether the observed behavior differences are statistically significant or potentially just a random result.

5.2.4 Parameter configuration

In order to retrieve the results for each case and algorithm, we need to configure the execution process and the algorithms accordingly. To be precise, all our results are retrieved from 30 independent algorithm executions to mitigate the influence of randomness. In the Stack case, we choose 5000, 10,000, 20,000 and 50,000 evaluations for Stack-A to Stack-D, respectively, and 20,000 for the Asc input model. In the CRA case, we choose 10,000, 20,000, 40,000 and 80,000 evaluations for CRA-A to CRA-D, respectively. Finally, in the transportation line case, we choose 25,000 evaluations for the creation of transportation lines from scratch and 5000 evaluations for the reconfiguration of transportation lines.

For all evolutionary algorithms, we choose the population size to be 100. Furthermore, we need to specify the evolutionary operators. As a selection operator, we use deterministic tournament selection with $n = 2$. Deterministic tournament selection takes n random candidate solutions from the population and selects the best one. The selected solutions are then considered for recombination. As recombination operator, we use the one-point crossover for all algorithms. The one-point crossover operator splits two parent solutions, i.e., orchestrated rule sequences, at a random position and merges them crosswise, resulting in two, new offspring solutions. The underlying assumption here is that traits which make the selected solutions fitter than other solutions will be inherited by the newly created solutions. Finally, we use a mutation operator to introduce slight, random changes into the solution candidates to guide the search into areas of the search space that would not be reachable through recombination alone. Specifically, we use our own mutation operator that exchanges one rule application at a random position with another with a mutation rate of five percent.

For the iterated local search, we need to configure how solutions are initialized and the parameters of the local search. We initialize by pre-applying a Hill Climbing-like population-based local search and taking the best w.r.t. a certain objective aggregation function. We define the tabu size, i.e., variable indices that may not be changed, to be 1. The subalgorithm is performed for $n = \sqrt{e \cdot p}$ evaluations with e being the total number of evaluations and p the population size.

Listing 8: Transportation Line Optimization Case - All Objectives

```
1  fitness = {
2    objectives = {
3      let producedItems = DeliveryStore.allInstances().storedItems->select(quality>0.5)->size();
4      let energy = (Component.allInstances()->collect(passiveEnergy+componentInfo.utilization*(
       ↪activeEnergy-passiveEnergy))->sum()/producedItems;
5      itemsProduced: maximize {producedItems}
6      wasteProduced: minimize {Items.allInstances()->select(quality < 0.5)->size()}
7      itemQuality: minimize{let items = DeliveryStore.allInstances().storedItems in items.quality->collect
       ↪(x|1.0-x)->sum()/items->size()}
8      energyConsumption: minimize{energy}
9      itemCosts: minimize {(20000+Component.allInstances().cost->sum()+energy*28+Component.allInstances().
       ↪componentInfo.productionCost->sum())/producedItems} //28=energy costs * time
10   }
11 }
```

For *Simulated Annealing*, we need to specify the temperature and temperature decrease (cf. Sect. 4.2). We choose $t_s = 0.2$ as inverse start temperature and $t_d = 0.007$ as temperature decrease value. We enforce a restart when the current solution cannot be improved after $c_c = 50$ steps and when the best solution cannot be improved after $c_b = 1000$ steps, i.e., it is never reset within a subalgorithm execution. For *Hill Climbing*, we specify the neighborhood size to be $n = 10$, always choose the best neighbor and restart the search after 10 unsuccessful improvement attempts.

5.3 Result analysis

This section describes the results of the experiments conducted with the three case studies. Based on the obtained results, we discuss the answers to our research questions.

5.3.1 Results for RQ1

Table 3 summarizes the minimum, average and maximum hypervolume values gained by executing the Stack case with three evolutionary algorithms (ϵ-MOEA, NSGA-II, NSGA-III), three local-search algorithms in single and iterated fashion (Hill Climbing: HC-sing./HC-iter., Random Descent: RD-sing./RD-iter. and Simulated Annealing: SA-sing./SA-iter.) and Random Search (Random S.) for the five input models (cf. Table 1). In the table, we have highlighted in bold the values of the best solutions.

The hypervolume indicator for the smallest stack example is always zero because this example can be solved by just a single rule application, so the Pareto front consists of only two solutions with normalized values (0, 1) and (1, 0). In that case, this indicator always returns zero. In any case, all algorithms deliver a good solution.

For larger problems, the results show that iterated Hill Climbing and random search are not suitable algorithms for this kind of problem. They do not only clearly take longer than the other algorithms, but also perform worse. The performance of local searches, especially Hill Climbing, is strongly tied to the problem structure. If the load can be balanced by just shifting stacks to the direct neighbor, Hill Climbing performs well. For all the randomly generated examples, *Stack-A* to *Stack-D*, this is the case to a certain extent. If the loads have to be distributed to a stack that is far away and therefore shifting some load to a direct neighbor does not solve the problem, such as in the *Stack-Asc* model, then local searches perform clearly worse than global searches. In fact, for *Stack-D*, single Hill Climbing performs statistically significantly better than Simulated Annealing. The numbers obtained for Simulated Annealing look like it performs better than the evolutionary algorithms, but not at the level of statistical significance. In contrast, this is exactly the other way around for *Stack-Asc*. There, the evolutionary algorithms perform better than Sim-

Table 3 Hypervolume and average runtime of the stack case

Case	Alg.	Hypervolume			Time (s)
		Min	Avg	Max	
Stack-A	ϵ-MOEA	0.000	0.000	0.000	**0**
	NSGA-II	0.000	0.000	0.000	0
	NSGA-III	0.000	0.000	0.000	0
	HC-iter.	0.000	0.000	0.000	1
	HC-sing.	0.000	0.000	0.000	0
	RD-iter.	0.000	0.000	0.000	0
	RD-sing.	0.000	0.000	0.000	0
	SA-iter.	0.000	0.000	0.000	1
	SA-sing.	0.000	0.000	0.000	1
	Random S.	0.000	0.000	0.000	1.21
Stack-B	ϵ-MOEA	0.083	0.424	**0.542**	**1.16**
	NSGA-II	0.500	0.540	**0.542**	1.76
	NSGA-III	0.500	0.540	**0.542**	1.81
	HC-iter.	0.000	0.097	0.500	5.03
	HC-sing.	**0.542**	**0.542**	**0.542**	2.96
	RD-iter.	0.125	0.412	**0.542**	3.43
	RD-sing.	0.125	0.429	**0.542**	3.51
	SA-iter.	0.292	0.533	**0.542**	3.43
	SA-sing.	**0.542**	**0.542**	**0.542**	3.31
	Random S.	0.000	0.000	0.000	8.65
Stack-C	ϵ-MOEA	0.459	0.564	**0.635**	**1.64**
	NSGA-II	0.553	0.622	**0.635**	2.52
	NSGA-III	0.553	0.610	**0.635**	2.58
	HC-iter.	0.000	0.049	0.424	10.5
	HC-sing.	0.553	0.606	0.624	3.86
	RD-iter.	0.235	0.466	0.588	5.04
	RD-sing.	0.235	0.449	0.600	5.05
	SA-iter.	0.541	0.623	**0.635**	5.95
	SA-sing.	**0.612**	**0.627**	**0.635**	5.86
	Random S.	0.000	0.000	0.000	20.4
Stack-D	ϵ-MOEA	0.373	0.566	0.690	**12.3**
	NSGA-II	0.509	0.606	**0.699**	16.5
	NSGA-III	0.450	0.599	0.684	15.8
	HC-iter.	0.000	0.018	0.210	75.2
	HC-sing.	**0.615**	**0.672**	0.694	30.0
	RD-iter.	0.324	0.498	0.637	40.3
	RD-sing.	0.419	0.508	0.595	42.4
	SA-iter.	0.577	0.626	0.656	34.7
	SA-sing.	0.588	0.625	0.657	34.1
	Random S.	0.000	0.000	0.000	142
Stack-Asc	ϵ-MOEA	0.000	0.281	**0.400**	3.72
	NSGA-II	**0.237**	**0.356**	**0.400**	5.80
	NSGA-III	0.225	0.345	**0.400**	6.06
	HC-iter.	0.000	0.042	0.263	17.7
	HC-sing.	0.000	0.000	0.000	**2.25**

Table 3 continued

Case	Alg.	Hypervolume			Time (s)
		Min	Avg	Max	
	RD-iter.	0.000	0.000	0.000	2.29
	RD-sing.	0.000	0.000	0.000	2.32
	SA-iter.	0.125	0.193	0.287	5.70
	SA-sing.	0.125	0.195	0.237	5.55
	Random S.	0.000	0.000	0.000	36.6

ulated Annealing and Simulated Annealing performs better than Hill Climbing. Here, both results are statistically significant. This shows that even for very simple problems, the concrete input model may have a great impact on which algorithm performs best.

Table 4 shows the hypervolume values for the Class Responsibility Assignment case. Overall, ϵ-MOEA performs best. It works significantly better than all the other algorithms in the largest example and is not worse in any other. As coupling and cohesion are strongly conflicting, Hill Climbing is not able to find meaningful results. Iterated Hill Climbing performs slightly, but statistically significantly, better as it tries out different search directions and starting points and thus avoids not being able to continue at all. Simulated Annealing works better than Hill Climbing, but worse than genetic algorithms. It is able to escape local optima, but cannot find vastly different results. For larger models, its efficiency compared to the evolutionary algorithms decreases. Iterated Simulated Annealing performs statistically significantly worse than non-iterated Simulated Annealing.

Table 5 shows the hypervolumes of the Transportation Line case. Since really low values are reported due to the limited solution variability, this indicator is not very meaningful. Thus, we also list the Inverted Generational Distance (IGD). Initially, we expected that genetic algorithms would perform best in this case. However, this is not the case.

For the transportation line creation, single Simulated Annealing and even random search outperform the evolutionary algorithms in terms of Inverted Generational Distance (IGD). Interestingly, all evolutionary search algorithms have the same maximum IGD, i.e., fail to find even a single solution of a product line actually producing products. A manual inspection of the solutions generated by the genetic algorithms showed that they often contain unnecessary machines. While there do exist many potentially working transportation lines, they are not that dense in the search space, which might be problematic for an algorithm dedicated to finding well-spread solutions. For the reconfiguration scenarios, the results are more evenly distributed since all solutions start with a working example. However, the evolutionary algorithms perform better in this case compared to the local ones.

Table 4 Hypervolume and average runtime of the CRA case

Case	Alg.	Hypervolume			Time
		Min	Avg	Max	
CRA-A	ϵ-MOEA	**0.421**	**0.421**	**0.421**	1 s
	NSGA-II	**0.421**	**0.421**	**0.421**	1 s
	NSGA-III	0.403	0.408	**0.421**	1 s
	HC-iter.	0.000	0.164	0.278	1 s
	HC-sing.	0.000	0.000	.000	**667 ns**
	SA-iter.	0.241	0.397	**0.421**	1.29 s
	SA-single	0.125	0.389	**0.421**	1.14 s
	Random S.	.000	.000	.000	1.84 s
CRA-B	ϵ-MOEA	0.461	0.465	0.471	3.44 s
	NSGA-II	**0.463**	**0.466**	**0.472**	3.73 s
	NSGA-III	0.426	0.446	0.456	3.53 s
	HC iter.	0.000	0.125	0.285	2.41 s
	HC-sing.	0.000	0.000	0.000	**0 s**
	SA-iter.	0.258	0.357	0.416	5.23 s
	SA-single	0.278	0.392	0.450	4.81 s
	Random S.	0.000	0.000	0.000	7.08 s
CRA-C	ϵ-MOEA	**0.618**	**0.642**	0.661	17.6 s
	NSGA-II	0.598	0.639	**0.661**	14.6 s
	NSGA-III	0.551	0.598	0.632	14.0 s
	HC-iter.	0.000	0.114	0.354	8.00 s
	HC-sing.	0.000	0.000	0.000	**0 s**
	SA-iter.	0.419	0.461	0.540	17.7 s
	SA-single	0.336	0.456	0.532	20.0 s
	Random S.	0.000	0.000	0.000	29.9 s
CRA-D	ϵ-MOEA	**0.493**	**0.558**	**0.621**	555 s
	NSGA-II	0.427	0.484	0.521	104 s
	NSGA-III	0.434	0.483	0.522	125 s
	HC-iter.	0.000	0.038	0.163	48.9 s
	HC-sing.	0.000	0.000	0.000	**0 s**
	SA-iter.	0.172	0.208	0.260	112 s
	SA-single	0.155	0.225	0.300	134 s
	Random S.	0.000	0.000	0.000	209 s

In Sect. 5.3.3, we describe some conclusions after the study realized.

5.3.2 Results for RQ2

The evaluation tables also show the average runtime for the investigated transformations for the different search algorithms and input models. A Mann–Whitney U-test on the performance shows that for the small models, such as CRA-A and the Stack models, evolutionary algorithms use statistically significantly less time than the other algorithms. This might be caused by copying solutions often for generating neighbors. This effect vanishes for larger models, where Hill Climbing performs better. This may be attributed to the fact

Table 5 Hypervolume, inverse generational distance and average runtime of the transportation line case

Case	Alg.	Hypervolume			IGD			Time (s)
		Min	Avg	Max	Min	Avg	Max	
Create generic	ϵ-MOEA	0.000	0.000	0.000	0.685	333 k	384 k	59.2
	NSGA-II	0.000	0.000	0.000	0.231	308 k	384 k	130
	NSGA-III	0.000	0.000	0.000	0.739	295 k	384 k	109
	Hill Climbing iterated	0.000	0.000	0.000	384 k	384 k	384 k	17.6
	Hill Climbing single	0.000	0.000	0.000	384 k	384 k	384 k	**1.02**
	RandomDescent single	0.000	0.000	0.000	384 k	384 k	384 k	1.88
	SA-iterated	0.000	0.000	0.000	0.539	38.4 k	384 k	186
	SA-single	0.000	**3.93 m**	**38.8 m**	**0.181**	**0.498**	1.178	84.0
	Random search	0.000	0.000	0.000	0.641	0.868	**1.048**	119
Create specific	ϵ-MOEA	0.000	1.04 m	11.3 m	0.948	151 k	302 k	92.0
	NSGA-II	0.000	3.02 m	11.3 m	0.862	90.5 k	302 k	162
	NSGA-III	0.000	2.51 m	11.3 m	0.809	101 k	302 k	131
	Hill Climbing iterated	0.000	2.12 m	**23.1 m**	0.976	271 k	302 k	26.0
	Hill Climbing single	0.000	0.000	0.000	302 k	302 k	302 k	**1.08**
	RandomDescent single	0.000	0.000	0.000	302 k	302 k	302 k	3.96
	SA-iterated	0.000	0.425 m	12.8 m	**0.428**	13.732	0 k	213
	SA-single	0.000	**4.03 m**	20.4 m	0.776	**0.943**	**1.295**	146
	Random search	0.000	0.000	0.000	1.448	1.559	1.638	201
Change generic	ϵ-MOEA	0.000	0.000	0.000	33.028	40.819	55.888	120
	NSGA-II	0.000	0.000	0.000	**1.118**	22.391	**33.028**	188
	NSGA-III	0.000	0.000	0.000	**1.118**	**19.375**	55.888	210
	Hill Climbing iterated	0.000	0.000	0.000	56.327	56.327	56.327	**91.8**
	Hill Climbing single	0.000	0.000	0.000	55.888	55.888	55.888	94.9
	RandomDescent single	0.000	0.000	0.000	55.888	55.888	55.888	93.9
	SA-iterated	0.000	0.000	0.000	33.048	33.226	33.574	458
	SA-single	0.000	0.000	0.000	33.029	33.371	33.542	118
	Random search	0.000	0.000	0.000	27.211	31.094	33.041	436
Change specific	ϵ-MOEA	0.000	0.000	0.000	1.249	**1.525**	1.908	128
	NSGA-II	0.000	0.000	0.000	**1.020**	1.599	1.889	153
	NSGA-III	0.000	0.000	0.000	1.359	1.711	**1.887**	147
	Hill Climbing iterated	0.000	0.000	0.000	2.813	2.813	2.813	**95.3**
	Hill Climbing single	0.000	0.000	0.000	2.791	2.791	2.791	98.5
	RandomDescent single	0.000	0.000	0.000	2.791	2.791	2.791	96.1
	SA-iterated	0.000	0.000	0.000	2.670	2.893	3.025	462
	SA-single	0.000	0.000	0.000	1.908	2.075	2.409	125
	Random search	0.000	0.000	0.000	1.345	1.851	2.759	564

that generating multiple neighbors is more efficient than generating a single neighbor in our implementation. The reason is that for generating an arbitrary number of neighbors, the execution of all transformations of a solution has to be conducted only once.

5.3.3 Lessons learned

From the gathered data presented in Sects. 5.3.1 and 5.3.2, we can draw the following conclusions.

First, for smaller examples, i.e., small input models, local-search algorithms perform reasonable well with respect to the search exploration. We have observed even cases where local-search algorithms perform better than global-search algorithms. In contrast, for bigger examples, global-search algorithms tend to outperform local-search algorithms. This, however, is also dependent on the problem structure. In particular, for problems that can be solved by following a particular solution path, local-search algorithms may perform better than global-search algorithms even for bigger

examples. For instance, we have observed in the Stack Load Balancing case that for already quite well balanced problem instances, local-search algorithms perform better than global-search algorithms.

Second, for problems where single changes of the solution can have a significant impact on the solution's fitness, evolutionary search algorithms are not applicable. With the Transportation Line Creation case, we have studied a particular interesting instance of such a problem. Here, a single change in the investigated production line might break the whole production leading to a solution with very low fitness. We have observed that in such a case, Simulated Annealing has an advantage as it avoids going in a production-breaking path with high fitness decrease. This might be also related to the solution encoding that we chose for MOMoT, which is encoding the application of transformation rules representing single changes of the investigated problem. In future work, we may investigate whether encoding the states of a model instead of the changes of the model brings improvements of evolutionary algorithms for such cases.

Third, we observed that for small input models, global-search algorithms are significantly more performant in terms of search time. For bigger input models, local-search algorithms are as performant as global-search algorithms.

5.4 Threats to validity

In this subsection, we elaborate on several factors that may jeopardize the validity of our results. According to Wohlin et al. [72], there are four basic types of validity threats that can affect the validity of our study. We cover each of these in the following paragraphs.

5.4.1 Conclusion validity

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. We use stochastic algorithms which by their nature produce slightly different results with every algorithm run. To mitigate this threat, we perform our experiment based on 30 independent runs for each case and algorithm and analyze the obtained results statistically with the Mann–Whitney U test with a confidence level of 99% ($\alpha = 0.01$) to test if significant differences exist between the measurements for different treatments. This test makes no assumption that the data is normally distributed and is suitable for ordinal data, so we can be confident that the statistical relationships we observed are significant. Regarding evaluation times, they are measured using standard Java features, namely `System.nanoTime()`. In order to mitigate the threat of having the garbage collector consuming resources, we perform `System.gc()` before each execution.

5.4.2 Construct validity

Construct validity is concerned with the relationship between theory and what is observed. Most of what we measure in our experiments are standard metrics such as Hypervolume and Inverted Generational Distance that are widely accepted as good proxies for quality of search approaches. Furthermore, we have based our conclusions in three specific cases for which a set of specific local-search and global-search algorithms have been implemented in MOMoT. Even for each local-search algorithm, we could have implemented different parameters to guide the local search, such as how much investigation of the neighbors is performed, for instance. Should we have constructed our experiments with different cases and/or different algorithms, we may have obtained different results. To mitigate this threat, we have implemented the algorithms in MOMoT based on well-described algorithms [40].

5.4.3 Internal validity

There are several internal threats to validity that we would like to mention. For instance, even though the trial-and-error method we used to define the parameters of our search algorithms is one of the most used methods [21], other parameter settings might yield different results. In fact, parameter tuning of search algorithms is still considered an open research challenge. We mitigated the risk of choosing completely odd parameter values by comparing automatically generated sensible parameters found by a simple genetic algorithm to our chosen values. The search performed with these values provided results which were not as good as the manually found parameters. Also and as mentioned before, the transformation problems selected may have an influence in the results. We have considered both a two-objective problem, the stack example, where local search automatically fulfills one of the objectives, i.e., short transformation length, a multi-objective problem with three objectives and a many-objective problem with five objectives.

5.4.4 External validity

The first threat in this category is the limited number of transformations we have evaluated, which externally threatens the generalizability of our results. Our results are based on the three case studies, and we have tried to minimize this threat by choosing three problems from three completely different domains. In any case, additional experiments are necessary to confirm our results and increase the chance of generalizability.

Second, we focus on Henshin as graph transformation tool and did not consider other graph transformation tools such as VIATRA [69], Fujaba [53], and GReAT [2] to mention

just a few. Furthermore, there are many other model transformation languages which are similar or quite different to the graph transformation language used in Henshin, e.g., consider model migration languages such as Epsilon Flock [57] or COPE [36]. Furthermore, Henshin may be considered as an in-place model transformation approach, i.e., an input model is modified to produce the output model. By this, also out-place model transformations may be emulated. However, we have not considered such transformation emulations in our case study and leave this kind of transformations as subject for future work. Finally, in order for our approach and results to be generalized also to out-place model transformation languages, we aim to apply it also to out-place model transformation languages such as QVT-O [54], QVT-R [54], TGGs [62], ETL [48], and RubyTL [12].

6 Related work

With respect to the contribution of this paper, we discuss three main threads of related work. First, we elaborate on the application of search-based techniques for generating model transformations from examples, which has been the first application target of search-based techniques concerning model transformations. Second, we discuss approaches which apply search-based techniques to optimize models. Finally, we survey work of applying search-based techniques done in the related field of program transformation.

6.1 Search-based model transformation generation

An alternative approach to develop model transformations from scratch is to learn model transformations from existing transformation examples, i.e., input/output model pairs. This approach is called model transformation by example (MTBE) [41,68,71] and several dedicated approaches have been presented in the past. Because of the huge search space when searching for possible model transformations for a given set of input/output model pairs, search-based techniques have been applied to automate this complex task [5,24,42–44,61]. While MTBE approaches do not foresee the existence of model transformation rules, on the contrary, the goal is to produce such rules, we discussed in this paper the orthogonal problem of finding the best sequence of rule applications for a given set of transformation rules in combination with transformation goals. Furthermore, MTBE approaches are mostly concerned with out-place transformations, i.e., generating a new model from scratch based on input models, while we focussed in this paper on in-place transformations, i.e., rewriting input models to output models. Finally, the authors in [63] propose the use of SBSE in MDE for optimizing regression tests for model transformations. In particular, they use a multi-objective approach to generate test cases, in the form of models that are the input for testing updated transformations. In our work, we assume to have correct model transformation rules available as a prerequisite but foresee as a possible future work the inclusion of oracle functions for model transformations in the search process.

Searching for transformation rule applications with search-based optimization techniques for high-level change detection has been presented in [6]. In the scenario of high-level change detection, the input model and the output model are given as well as the possible transformation rules. The goal is to find the best sequence of rule applications which gives the most similar output model when applying the rule application sequence to the input model. In other words, the high-level change detection we have investigated previously is a special case which is now more generalized in the proposed framework by having the possibility to specify arbitrary goals for the search. Another combination of model engineering and SBSE is presented in [20]; however, in this framework the possible changes to the models are not defined as transformation rules, but are generally defined directly on the generic genotype representations of the models.

6.2 Transformation as search process

Searching for model transformation results is currently supported by approaches using some kind of constraint solver. For instance, Kleiner et al. [47] introduce an approach they call *transformation as search* where they use constraint programming to search and produce a set of target models from a given source model. Another approach is proposed by Gogolla et al. [33] where so-called transformation models are defined with OCL and then translated to a constraint solver to find valid output models for given input models. Compared to MOMoT, these approaches aim for a full enumerative approach where concrete bounds for constraining the search space have to be given. Furthermore, these approaches search for models fulfilling some correctness constraints, but finding optimal models based on some objectives is not natively supported in such approaches. In [18,19] an approach extending the QVT Relations language is presented which also foresees the inclusion of transformation goals in the transformation specifications including different transformation variants. However, the search for finding the most suitable transformation variant is not based on meta-heuristics but is delegated to the model engineers who have to make decisions for guiding the search process. In [59], the authors present an approach for representing the variability of model migrations which are necessary for dealing with meta-model/model co-evolution scenarios. The different migration alternatives are represented on the intentional level using feature models which allows an explicit visualization of conflicting solutions, i.e., conflicting decisions. While this approach allows

to capture the variability of the solution space, the evaluation of the solutions is currently not targeted.

Another very recent line of research concerning the search of transformation results is already applying search-based techniques to orchestrate the transformation rules to find models fulfilling some given objectives. The authors in [17] propose a strategy for integrating multiple single-solution search techniques directly into a model transformation approach. In particular, they apply exhaustive search, randomized search, Hill Climbing, and Simulated Annealing. Their goal is to explicitly model the search algorithms as graph transformations. Compared to this approach, our approach is going into the opposite direction. We are reusing already existing search algorithms provided by dedicated frameworks from the search-based optimization domain. The most related approach to MOMoT is presented by Abdeen et al. [1]. They also address the problem of finding optimal sequences of rule applications, but they focus on population-based search techniques. Thereby, they consider the multi-objective exploration of graph transformation systems, where they apply NSGA-II [16] to drive rule-based design space explorations of models. For this purpose, they have directly extended a model transformation engine to provide the necessary search capabilities.

Our presented work follows the same spirit as the previous mentioned two approaches; however, our aim is to provide a loosely coupled framework which is not targeted to a single optimization algorithm, but allows to (re)use the most appropriate one for a given transformation problem. Additionally, we aim to support the model engineer in using these algorithms through a dedicated configuration DSL and provide analysis capabilities to evaluate the performance of different algorithms. As an interesting line of future research, we consider the evaluation of the flexibility and performance of the different approaches we have now for combining MDE and SBSE: modeling the search algorithms as transformations [17], integrating the search algorithms into transformation engines [1], or combining transformation engines with search algorithm frameworks as we are doing with MOMoT.

Finally, we also contributed to the Transformation Tool Contest (TTC) 2016 a case study [27] about a search-based model transformation problem, which resulted in eight different solution submissions.[10] In particular, six out of eight solutions resorted on meta-heuristic algorithms for solving the given case study. Some approaches re-implemented genetic algorithms within the model transformation languages (UML-RSDS and Henshin), others combined the transformation engines with search-based algorithm implementations (VIATRA-DSE, ATL/Java, and MDEOptimiser), and one approach translated the complete transformation problem to a genetic algorithm framework (SIGMA).

In MOMoT, we are also following the strategy of combining the transformation engine with search-based algorithm implementations (by reusing an extended version of the MOEA framework). Compared with the other TTC solutions, we allow for a more flexible selection and combination of different meta-heuristic search algorithms.

6.3 Search-based program transformation

Program transformation is a field closely related to model transformation [70], and thus, similar problems are occurring in both fields. For instance, one challenging transformation in this field is the (re)-modularization of programs which has been tackled as a many-objective optimization problem in [52]. Another challenging program transformation scenario is to enhance the readability of source code given certain metrics. In this context, we are aware of a related approach that discusses the search-based transformation of programs [22,23]. In particular, a set of rewriting rules is presented to optimize the readability of the code and dedicated metrics are proposed and used as fitness function. As search techniques, random search, Hill Climbing, and genetic algorithms are used.

Our approach follows a similar idea of finding optimal sequences of rule applications, but in our case we are focussing on model structures and model transformations instead of source code. Furthermore, we also consider the application of different search algorithm for finding optimal rule sequences, and in addition, provide an evaluation for three cases. Moreover, we consider the instantiation of our framework for the problem of program transformation in combination with model-driven reverse engineering tools [10] as an interesting subject for future work to further evaluate our approach.

7 Conclusion and future work

In this paper, we have shown how to apply different meta-heuristic search algorithms for model transformation problems which are considered to be multi-objective optimization problems. With MOMoT, Henshin model transformation rule application sequences are computed to optimize the resulting output models. This is achieved by an adapted version of the MOEA framework in order to realize not only population-based recombination but also local searchers exploring the neighborhood. In addition, we also contribute a tool for the scientific community to perform experimental research focusing on the usage of different meta-heuristic search algorithms and their combination for MDE problems. MOMoT as

[10] http://www.transformation-tool-contest.eu/2016/solutions_cra.html.

well as the experiments presented in this paper are available as open-source.

MOMoT provides a wide spectrum of different search algorithms and their combination for orchestrating transformation rules, but as always, there is still room for improvements. First, as we currently provide different algorithms but not their combination, we plan to incorporate Memetic Algorithms which allow for combined usage of global and local searchers. Second, we would like to explore the combination of search-based and approximate model transformations [66], i.e., how much precision may be traded for performance. Third, we plan to investigate if MOMoT is also applicable to typical out-place transformation scenarios. Here, one of the most interesting research questions is how to formulate and evaluate the fitness of output models which may require the integration of simulation techniques.

Acknowledgements The work of Javier Troya is funded by the European Commission (FEDER) and the Spanish and the Andalusian R&D&I programmes under grants and projects BELI (TIN2015-70560-R) and COPAS (P12-TIC-1867). Finally, the work of Manuel Wimmer is funded by the Austrian Federal Ministry of Science, Research and Economy and the National Foundation for Research, Technology and Development as well as by the Austrian Science Fund (FWF): P 28519-N31.

References

1. Abdeen, H., Varró, D., Sahraoui, H., Nagy, A.S., Debreceni, C., Hegedüs, A., Horváth, A.: Multi-objective optimization in rule-based design space exploration. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE), pp. 289–300 (2014)
2. Agrawal, A.: Graph rewriting and transformation (GReAT): a solution for the model integrated computing (MIC) Bottleneck. In: Proceedings of the 18th International Conference on Automated Software Engineering (ASE'03), pp. 364–368 (2003)
3. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE), pp. 1–10 (2011)
4. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 121–135 (2010)
5. Baki, I., Sahraoui, H.A., Cobbaert, Q., Masson, P., Faunes, M.: Learning implicit and explicit control in model transformations by example. In: Proceedings of 17th International Conference on Model-Driven Engineering Languages and Systems (MODELS), pp. 636–652 (2014)
6. ben Fadhel, A., Kessentini, M., Langer, P., Wimmer, M.: Search-based detection of high-level model changes. In: Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM), pp. 212–221 (2012)
7. Biermann, E., Ermel, C., Taentzer, G.: Lifting parallel graph transformation concepts of model transformation based on the eclipse modeling framework. Electron. Commun. EASST **26**, 1–19 (2010)
8. Bowman, M., Briand, L., Labiche, Y.: Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. IEEE TSE **36**(6), 817–837 (2010)
9. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool, San Rafael (2012)
10. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: A generic and extensible framework for model driven reverse engineering. In: Proceedings of the 25th International Conference on Automated Software Engineering (ASE), pp. 173–174 (2010)
11. Cohen, W.: Machine learning proceedings 1994: Proceedings of the Eighth International Conference. Elsevier Science (2014)
12. Cuadrado, J.S., Molina, J.G., Tortosa, M.M.: RubyTL: A practical, extensible transformation language. In: Proceedings of the 2nd European Conference on Model Driven Architecture—Foundations and Applications (ECMDA-FA), pp. 158–172 (2006)
13. Darwin, C.: On the origin of species by means of natural selection. John Murray, London (1859)
14. Deb, K., Jain, H.: Handling many-objective problems using an improved NSGA-II procedure. In: Proceedings of the 7th World Congress on Evolutionary Computation (CEC), pp. 1–8 (2012)
15. Deb, K., Jain, H.: An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems with box constraints. IEEE Trans. Evol. Comput. **18**(4), 577–601 (2014)
16. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)
17. Denil, J., Jukss, M., Verbrugge, C., Vangheluwe, H.: Search-based model optimization using model transformations. In: Amyot, D., Fonseca i Casas, P., Mussbacher, G. (eds.) Proceedings of the 8th International Conference on System Analysis and Modeling (SAM), pp. 80–95 (2014)
18. Drago, M.L., Ghezzi, C., Mirandola, R.: Towards quality driven exploration of model transformation spaces. In: Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS'11), pp. 2–16 (2011)
19. Drago, M.L., Ghezzi, C., Mirandola, R.: A quality driven extension to the QVT-relations transformation language. Comput. Sci. R&D **30**(1), 1–20 (2015)
20. Efstathiou, D., Williams, J.R., Zschaler, S.: Crepe complete: multi-objective optimisation for your models. In: Proceedings of the First International Workshop on Combining Modelling with Search- and Example-Based Approaches (CMSEBA'14) @ MODELS, vol. 1340, pp. 25–34. CEUR-WS.org (2014)
21. Eiben, A.E., Smit, S.K.: Parameter tuning for configuring and analyzing evolutionary algorithms. Swarm Evol. Comput. **1**(1), 19–31 (2011)
22. Fatiregun, D., Harman, M., Hierons, R.M.: Search based transformations. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'03), *LNCS*, vol. 2724, pp. 2511–2512. Springer (2003). https://doi.org/10.1007/3-540-45110-2_154
23. Fatiregun, D., Harman, M., Hierons, R.M.: Evolving transformation sequences using genetic algorithms. In: Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), pp. 66–75 (2004)
24. Faunes, M., Sahraoui, H.A., Boukadoum, M.: Genetic-programming approach to learn model transformation rules from examples. In: Proceedings of the 6th International Conference on Theory and Practice of Model Transformations (ICMT), pp. 17–32 (2013)
25. Fleck, M., Troya, J., Wimmer, M.: Marrying search-based optimization and model transformation technology. In: Proceedings of the 1st North American Search Based Software Engineering Symposium (NasBASE) (2015)
26. Fleck, M., Troya, J., Wimmer, M.: Search-based model transformations with MOMoT. In: Proceedings of the 9th International

Conference on Theory and Practice of Model Transformations (ICMT), pp. 79–87 (2016)
27. Fleck, M., Troya, J., Wimmer, M.: The class responsibility assignment case. In: Proceedings of the 9th Transformation Tool Contest (TTC 2016), pp. 1–10 (2016)
28. Fleck, M., Troya, J., Wimmer, M.: Search-based model transformations. J. Softw. Evol. Process **28**(12), 1081–1117 (2016)
29. Fleck, M., Troya, J., Kessentini, M., Wimmer, M., Alkhazi, B.: Model transformation modularization as a many-objective optimization problem. IEEE Trans. Softw. Eng (2017). https://doi.org/10.1109/TSE.2017.2654255
30. Fogel, L.J.: Toward inductive inference automata. In: Proceedings of the 2nd International Federation for Information Processing (IFIP), pp. 395–399 (1962)
31. Fogel, L.J.: Intelligence Through Simulated Evolution. Wiley, New York (1966)
32. Glover, F.: Future paths for integer programming and links to artificial intelligence. Comput. Operat. Res **13**(5), 533–549 (1986)
33. Gogolla, M., Hamann, L., Hilken, F.: On static and dynamic analysis of UML and OCL transformation models. In: Proceedings of the Workshop on Analysis of Model Transformations (AMT) @ MODELS, CEUR Workshop Proceedings, vol. 1277, pp. 24–33. CEUR-WS.org (2014)
34. Goldberg, D.E., Lingle Jr., R.: Alleles, loci, and the traveling salesman problem. In: Proceedings of the 1st International Conference on Genetic Algorithms (ICGA), pp. 154–159 (1985)
35. Harman, M.: The current state and future of search based software engineering. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 342–357 (2007)
36. Herrmannsdoerfer, M.: COPE—a workbench for the coupled evolution of metamodels and models. In: Proceedings of the 3rd International Conference on Software Language Engineering (SLE), pp. 286–295 (2010)
37. Holland, J.H.: Outline for a logical theory of adaptive systems. J. ACM **9**(3), 297–314 (1962)
38. Holland, J.H.: Adaptation in Natural and Artificial Systems. MIT Press, Cambridge (1975)
39. Holland, J.H.: Adaptation in Natural and Artificial Systems. MIT Press, Cambridge (1992)
40. Ishibuchi, H., Kaige, S.: Implementation of simple multiobjective memetic algorithms and its applications to knapsack problems. Int. J. Hybrid Intell. Syst. **1**(1), 22–35 (2004)
41. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model transformation by-example: a survey of the first wave. In: Conceptual Modelling and Its Theoretical Foundations. LNCS, vol. 7260, pp. 197–215. Springer, Heidelberg (2012)
42. Kessentini, M., Sahraoui, H.A., Boukadoum, M.: Model transformation as an optimization problem. In: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS'08), LNCS, vol. 5301, pp. 159–173. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-87875-9_12
43. Kessentini, M., Bouchoucha, A., Sahraoui, H.A., Boukadoum, M.: Example-based sequence diagrams to colored petri nets transformation using heuristic search. In: Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA), pp. 156–172 (2010)
44. Kessentini, M., Sahraoui, H.A., Boukadoum, M., Benomar, O.: Search-based model transformation by example. Softw. Syst. Model. **11**(2), 209–226 (2012)
45. Kessentini, M., Langer, P., Wimmer, M.: Searching models, modeling search: on the synergies of SBSE and MDE. In: Proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE) @ ICSE, pp. 51–54 (2013)
46. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220**(4598), 671–680 (1983)
47. Kleiner, M., Didonet Del Fabro, M., Queiroz Santos, D.: Transformation as search. In: Proceedings of the 9th European Conference on Modelling Foundations and Applications (ECMFA), pp. 54–69 (2013)
48. Kolovos, D.S., Paige, R.F., Polack, F.: The epsilon transformation language. In: Proceedings of the 1st International Conference on Theory and Practice of Model Transformations (ICMT), pp. 46–60 (2008)
49. Koza, J.R.: Genetic Programming. MIT Press, Cambridge (1992)
50. Mann, H.B., Whitney, D.R.: On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Stat. **18**(1), 50–60 (1947)
51. Masoud, H., Jalili, S.: A clustering-based model for class responsibility assignment problem in object-oriented analysis. JSS **93**, 110–131 (2014)
52. Mkaouer, W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K., Ouni, A.: Many-objective software remodularization using NSGA-III. ACM Trans. Softw. Eng. Methodol. **24**(3), 17:1–17:45 (2015)
53. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE), pp. 742–745 (2000)
54. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1 (2011). http://www.omg.org/spec/QVT/1.1/
55. Rechenberg, I.: Cybernetic solution path of an experimental problem. Library Translation 1112, Royal Aircraft Establishment (1965)
56. Rechenberg, I.: Evolutionsstrategie: Optimierung Technischer Systeme Nach Prinzipien der Biologischen Evolution. Frommann-Holzboog, Stuttgart (1973)
57. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model migration with epsilon flock. In: Proceedings of the 3rd International Conference on Theory and Practice of Model Transformations (ICMT), pp. 184–198 (2010)
58. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. **14**(2), 131–164 (2009)
59. Ruscio, D.D., Etzlstorfer, J., Iovino, L., Pierantonio, A., Schwinger, W.: Supporting variability exploration and resolution during model migration. In: Proceedings of the 12th European Conference on Modelling Foundations and Applications (ECMFA), pp. 231–246 (2016)
60. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice-Hall, Upper Saddle (2009)
61. Saada, H., Huchard, M., Nebut, C., Sahraoui, H.A.: Recovering model transformation traces using multi-objective optimization. In: Proceedings of the 28th International Conference on Automated Software Engineering (ASE), pp. 688–693 (2013)
62. Schürr, A.: Specification of graph translators with triple graph grammars. In: Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16–18, 1994, Proceedings, pp. 151–163 (1994)
63. Shelburg, J., Kessentini, M., Tauritz, D.: Regression testing for model transformations: a multi-objective approach. In: Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE), pp. 209–223 (2013)
64. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional, Reading (2008)
65. Talbi, E.G.: Metaheuristics: From Design to Implementation. Wiley Publishing, New York (2009)
66. Troya, J., Wimmer, M., Burgueño, L., Vallecillo, A.: Towards approximate model transformations. In: Proceedings of the Work-

shop on Analysis of Model Transformations (AMT) @ MODELS, pp. 44–53 (2014)
67. Van Laarhoven, P.J., Aarts, E.H.: Simulated annealing. In: Simulated Annealing: Theory and Applications. Springer, Berlin (1987)
68. Varró, D.: Model transformation by example. In: Proceedings of 9th International Conference on Model-Driven Engineering Languages and Systems (MODELS), pp. 410–424 (2006)
69. Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z.: Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. Softw. Syst. Model. **15**(3), 609–629 (2016)
70. Visser, E.: A survey of rewriting strategies in program transformation systems. Electron. Note. Theor. Comput. Sci. **57**(2), 109–143 (2001)
71. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards model transformation generation by-example. In: Proceedings of the 40th Hawaii International Conference on Systems Science (HICSS), p. 285b (2007)
72. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B.: Experimentation in Software Engineering. Springer, Berlin (2012)
73. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the strength pareto evolutionary algorithm. TIK-Report 103, Computer Engineering and Networks Laboratory (TIK) and Swiss Federal Institute of Technology (ETH) Zurich (2001)
74. Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C.M., da Fonseca, V.G.: Performance assessment of multiobjective optimizers: an analysis and review. IEEE Trans. Evol. Comput. **7**(2), 117–132 (2003)