Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

NASBASE 00 (2015) 1–16

NASBASE

www.elsevier.com/locate/procedia

Marrying Search-based Optimization and Model Transformation Technology

Martin Fleck, Javier Troya, Manuel Wimmer

Business Informatics Group, Vienna University of Technology, Austria

Abstract

Model transformations are an important cornerstone of model-driven engineering. Thus, various model transformation languages have been proposed, most of which follow the rule-based paradigm. The application of transformation rules is realized either following the apply-as-long-as-possible strategy or an orchestration of the rules has to be provided. However, two main limitations arise from following these two approaches. First, the goals of the transformations are implicitly hidden in the rule encoding and their orchestration specifications. Second, manually finding the best rule orchestration for a particular scenario is a complex problem due to the high number of rule combination possibilities.

To tackle these limitations, we present in this paper a novel framework which builds on the non-intrusive integration of optimization and model transformation technology. This integration allows for search-based exploration of rule applications and to make the goals of transformations explicit. In particular, we formulate the rule application problem as an optimization problem. The proposed framework provides several algorithms for local and global searches of rule applications guided by single and multiple objectives expressed in terms of models. We present different instantiations of our framework to demonstrate its feasibility and benefits by several case studies in the field of software engineering.

Keywords: Model transformation; Rule orchestration; Model-driven Software Engineering; Search-based Software Engineering

1. Introduction

Transformations are an important concept in computer science in general and in software engineering in particular, since indeed, computation can be viewed as data transformation. The same situation occurs in Model-Driven Engineering (MDE) [1], which has model transformations at its heart [2]. In MDE, models are the central artifacts which describe complex systems from various viewpoints and at multiple levels of abstraction using appropriate modeling formalisms. Model transformations provide the essential mechanisms for manipulating and transforming models, e.g., abstracting software models such as class models from existing source code using reverse engineering techniques [3]. In fact, several distinct categories of model transformations have been identified [4, 5]. In broader terms, there exist declarative, imperative, and hybrid approaches. In any case, most of them are expressed by means of transformation rules.

A crucial aspect when dealing with model transformations is the orchestration of the rules that compose them, i.e., the order in which they are executed. This orchestration can be defined implicitly or explicitly [4]. In an implicit orchestration, the developer has no control over the order in which rules are triggered. This task is delegated to the transformation engine. This is typically the case with purely declarative languages, such as QVT Relations [6] and many graph transformation languages. Other languages offer mechanisms to explicitly define the set of rule orchestrations. For instance, ATL [7] is a hybrid transformation language that offers the possibility of partially orchestrating the rules that compose a model transformation by explicitly making calls to so-called lazy rules in the declarative part. Other languages provide more dedicated mechanisms to schedule rules, such as VIATRA [8], which offers rule scheduling using abstract state machines.

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 suffering from two major drawbacks. First, the effect a rule application has on the characteristics of the resulting model is implicitly hidden in the behavior encoded by the rule and may depend on previously applied rules. In fact, a given rule may check for some information produced by another rule. Second, the number of rule combinations may be very large or even infinite, especially when considering the input parameters a rule may have, making a manual exploration of rule orchestrations very difficult.

Based on ideas that we have initially outlined in previous work [9] on combining Search-Based Software Engineering (SBSE) [10] and MDE, in this paper we consider the problem of finding the best orchestration for 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 [11], 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. 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.

In this paper, we propose an algorithm-agnostic approach called *MOMoT* (Marrying Optimization and Model Transformations) to encode model transformations as optimization problems and provide a loosely coupled framework bridging two existing and well-established base frameworks. Our framework is developed within the Eclipse Modeling Framework (EMF)[1] and builds upon Henshin[2] [12] to define model transformations and the MOEA framework[3] for providing optimization techniques. Henshin is a graph transformation engine that provides a graphical notation for defining model transformations as graph transformation rules. The MOEA framework provides several multi-objective evolutionary algorithms, such as NSGA-II [13], NSGA-III [14], and ε-MOEA [15], as well as tools to execute and statistically test these algorithms. In addition, our framework integrates other optimization techniques, including for instance single-objective and local search techniques, and provides hooks for integrating further techniques.

The remainder of the paper is structured as follows. In Section 2 we give a short introduction into model transformation and the Henshin framework. Section 3 introduces our approach and describes how the framework has been developed, in particular, which techniques we provide and how the framework can be extended. Section 4 describes the evaluation of our framework, before we give an overview on related work in Section 5. Finally, Section 6 concludes the paper with an outlook on future work.

2. Prerequisites: Models, Meta-models, and Model Transformations

In this section, we give a short introduction to model transformation based on an example.

2.1. Model Transformation

In the MDE field, there are many different model transformation kinds [4] such as *model-to-model*, *text-to-model*, and *model-to-text* transformations. A model transformation can further be categorized as *out-place*

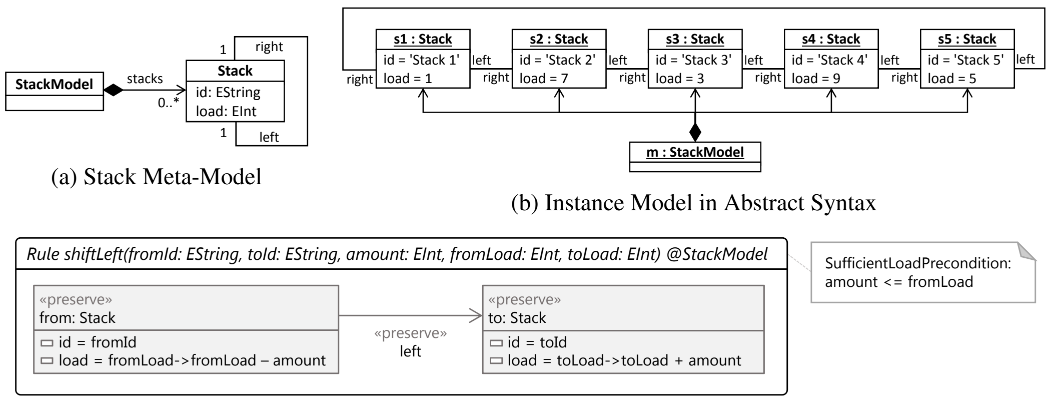
[1]`http://www.eclipse.org/modeling`
[2]`http://www.eclipse.org/henshin`
[3]`http://www.moeaframework.org`

if it creates new models from scratch or as *in-place* if it rewrites the input models until the output models are obtained. In this paper we focus on in-place model-to-model transformations, which can be expressed using graph transformation rules. The applicability of graph transformations for model transformations rests upon the fact that most models exhibit a graph-based structure, e.g., consider the underlying structure of UML class diagrams or state machines, whereas the meta-models of the models act as type graphs. The initial graph representing a model evolves through the application of graph transformation rules until the execution stops and we obtain the output graph, i.e., the output model. There is a plethora of frameworks and languages to define these kinds of transformations, such as Henshin [12], AGG [16], Maude [17], AToM^3 [18], e-Motions [19], and VIATRA [8].

Henshin. In this paper we use Henshin [12] to instantiate our approach, since it offers a rich language and associated tool set for in-place transformations of Ecore-based models. Ecore is the central meta-language in EMF that defines the concepts that can be used to create modeling languages. However, please note that our approach is not conceptually limited to Henshin. Henshin comes along with a powerful declarative model transformation language that has its roots in attributed graph transformations and offers the possibility for formal reasoning. It also provides the concept of transformation units to define control structures for rule applications in a modular way. Transformation rules are considered a special kind of transformation unit, which can have input parameters similar to what is known from programming, e.g., references to model elements or primitive values. In Henshin, as well as in many other graph transformation languages, a transformation rule consists of left- and right-hand side graphs, which describe model patterns to be matched and changes to be applied, respectively. Rules may also have positive and negative application conditions (PACs and NACs), which specify the mandatory presence and absence of graph patterns before the rule may be applied, respectively. Graphs are attributed, and nodes, edges, and attributes refer to *EClass*, *EReference* and *EAttribute* classes of the Ecore meta-model. Other transformation units provide mechanisms to orchestrate these rules, e.g., sequential units, priority units or amalgamation units [20]. Due to the different possibilities of orchestrating rules in Henshin, a mechanism that automatically determines the optimal execution order of rules for a specific scenario may ease the work of the transformation designer substantially.

2.2. Transformation Example

The running example we use to describe our approach in this paper 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 Figure 1a. 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 instance of this meta-model composed of five stacks with different loads is shown in Figure 1b.



(a) Stack Meta-Model

(b) Instance Model in Abstract Syntax

(c) *ShiftLeft* Rule to shift load

Fig. 1: Stack System

The figure depicts the abstract model syntax, i.e., the graph-based representation of the model information modulo its notation. To manipulate this instance model 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 Figure 1c—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*, *fromLoad*, and *toLoad* can be set. However, how much load should be shifted (*amount*) requires input from the user. 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.

2.3. *Exemplary Transformation Execution Scenario*

Let us assume we have a scenario where the goal of executing the transformation is to have a model where the load among the stacks is equally distributed, i.e., a minimal standard deviation of the stack loads. In our example (cf. Figure 1b) that would mean that each stack should have a load of five. Without defining control structures for the rule applications, an as-long-as-possible execution of the rules would be exchanging parts indefinitely because we can not specify a termination criteria in Henshin. Therefore we have to derive a rule orchestration by hand. Assuming we do not know the standard deviation in advance, one possible rule orchestration retrieved through trial-and-error is shown in Listing 1. Each rule application is represented as a function call, the function name corresponding to the rule name and the arguments matching the rule parameters shown in Figure 1c. However, there are many different rule orchestrations that lead to the same result and the one we have found may not be the shortest one. Therefore, additional support in exploring different rule orchestrations is needed. Our approach to provide this support is explained in the next section.

Listing 1: Textual representation of a solution for the Stack example

```
{ ShiftLeft ('Stack_4', 'Stack_3', 2, 9, 3), ShiftLeft ('Stack_2', 'Stack_1', 3, 7, 1),
  ShiftRight('Stack_5', 'Stack_1', 1, 5, 4), ShiftRight('Stack_4', 'Stack_5', 1, 7, 4),
  ShiftLeft ('Stack_3', 'Stack_2', 1, 5, 4), ShiftLeft ('Stack_4', 'Stack_3', 1, 6, 4) }
```

3. Search-based Rule Application with MOMoT

This section gives an overview on the MOMoT approach and its required inputs and provided outputs.

3.1. *MOMoT at a Glance*

One way to efficiently explore the potentially huge rule orchestration state space of a problem domain, which may even be infinite and multi-modal, is to apply SBSE techniques within model transformation systems. We therefore propose a loosely coupled framework that provides a bridge between two types of existing frameworks: one to encode model transformation problems and one for search-based optimization. Reusing the existing functionality of these base frameworks as much as possible is the central principle of our framework. This avoids the necessity for users to learn new formalisms and reduces the risk of introducing additional errors through re-implementation. Furthermore, there is little to no delay in receiving updates for bug-fixes, new functions, algorithms, or optimizations from the two existing frameworks.

As mentioned in the previous section, in this paper we build upon Henshin which supports model transformations of Ecore-based models and the MOEA framework which provides a set of multi-objective evolutionary algorithms with additional analytical performance measures and which can be easily extended with new algorithms. While in the rest of the paper we discuss our framework in the light of these two base frameworks, the approach itself is generic so that other frameworks may also be used.

An overview of our approach is depicted in Figure 2. Instead of manually deriving an orchestration of transformation rules for a given scenario in the specific problem domain, dedicated search algorithms are employed to calculate the orchestration of the rules based on a given set of objectives and constraints.

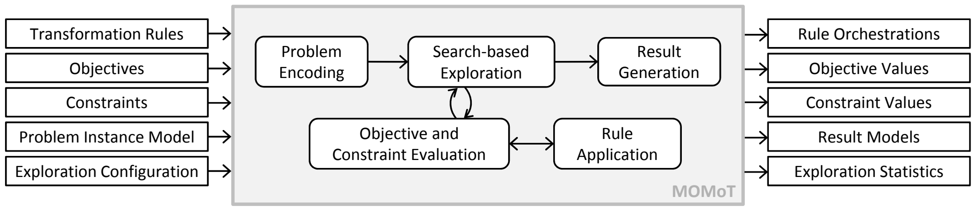


Fig. 2: Overview of the MOMoT approach

To realize our approach, we need the following ingredients: (*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, and (*iii*) a random solution generator that is used for the generation of an initial, random individual or random population by many search-based algorithms. To further support the use of multi-objective evolutionary algorithms, we additionally provide (*iv*) generic objectives and constraints for our solution encoding, and (*v*) generic mutation operators that can modify the respective solutions. For supporting local search algorithms, we provide a base interface and implementation as well as different neighborhood functions. In the following sections, we describe the different parts of our approach on the basis of the Stack example introduced in the previous section.

3.2. *Problem encoding*

As typical in model-driven engineering, the problem domain itself is defined as a meta-model representing a modeling language (cf. the meta-model of the Stack problem depicted in Figure 1a). Based on the specific problem domain, a user can define both concrete problem instances (cf. Figure 1b) and transformation rules that specify how problem instances can be modified in order to produce a solution (cf. Figure 1c).

Objectives and Constraints. The quality of each solution candidate is defined by a fitness function that evaluates multiple objective 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. In our approach, we can distinguish between objectives that are problem domain-specific, e.g., minimizing the standard deviation of the loads in a Stack domain, and objectives that relate to the solution encoding, e.g, minimizing the number of transformations that should be applied to achieve a solution. An objective can be defined either by providing a specific Java method returning the respective objective value or by specifying model queries in the Object Constraint Language (OCL). Listing 2 shows the OCL query for retrieving the standard deviation of the loads in the Stack domain. OCL is a standardized and formal language to describe expressions, constraints and queries on models that can be automatically evaluated by using Eclipse OCL. Additionally, a solution candidate may be subjected to a number of constraints in order for the solution to be valid. Depending on the algorithm, invalid solutions may be filtered out completely or may receive a low ranking in relation to the magnitude of the constraint violation. As with objectives, we distinguish between domain-specific constraints, e.g., a stack may never have a negative load, and solution-specific constraints, e.g., a specific transformation rule has to be applied. Again, we can leverage OCL to specify constraints for the problem domain or provide a Java method returning the magnitude of the constraint violation. However, as we are dealing with a graph transformation system, we can also make use of the expressive power of the graph transformation engine itself by specifying NACs directly in the rules. By doing so, we can effectively avoid the generation of invalid solution candidates, resulting in a potentially smaller search space. However, due to the cost of graph pattern matching, the application of NACs may also introduce an additional overhead, depending on the NACs complexity and the number of pruned solutions.

Listing 2: OCL query on standard deviation of stacks

```
package stack      -- complement the declarations of the stack package described by the meta-model
context StackModel -- subsequent model elements are defined here
  -- attribute definitons
  def: LoadSum: Real = stacks->collect(load)->sum()
  def: NrStacks: Integer = stacks->size()
  def: MeanLoad: Real = LoadSum / NrStacks
  def: DeviationSum : Real = stacks->collect((load-MeanLoad)*(load-MeanLoad))->sum()
  def: Variance : Real = DeviationSum / NrStacks
  -- query
  Variance.sqrt() -- square root function may be defined externally
endpackage
```

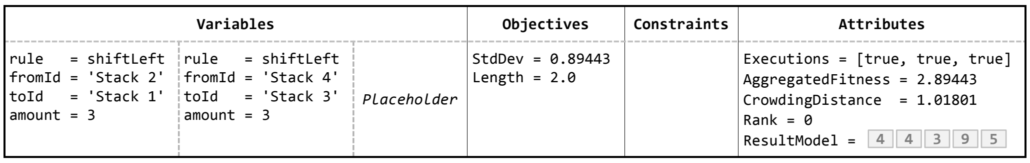
In summary, the problem domain is represented as a modeling language, from which a concrete problem instance model is created. Transformation rules defined upon the language can modify the model instances. The objectives that should be achieved may be defined via model queries in OCL. The constraints that a possible solution must fulfill can also be specified in OCL, but may also be encoded as NACs directly in the rules.

3.3. Solution representation

A solution in general consists of a number 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. Additional information about a solution such as the solution rank assigned by some algorithms may be stored in the attributes of that solution, which serve as a key-value storage. As we deal with a transformation problem, there are two common ways in representing 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. Therefore, a decision variable in our solution is one rule application. A rule application refers to one specific transformation rule plus the values for all parameters of that rule that make the rule applicable. As a special case, we also allow the use of rule application *placeholders*, i.e., rule applications that are not actually executed and do not have any effect. This allows the actual solution length to vary in cases where the solution length must be fixed. An example Stack solution with a sequence of two rule applications and one placeholder is depicted in Figure 3.

Parameters. When dealing with model transformations, we can distinguish between two kinds of rule 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 changing properties. In the Stack example, the only user parameter is the *amount* of load that should be shifted from one stack to another. When applying a rule manually, a user can provide a value for such a parameter either programmatically or via a dedicated user interface.

Random Solution Generation. To create random solutions with a high variance of parameter values to cover as much area of the search space as possible, we provide random parameter value generators for most primitive values. By default, the range of these values is the range of the data type of the respective parameter, e.g., for Integer in Java the value can range from -2^{31} to $2^{31} - 1$. Although an efficient search algorithm should quickly remove values that are not beneficial in a specific scenario, the user may restrict this range as part of the exploration configurations (cf. Section 3.4) to prune such unfruitful areas of the search space in advance. Furthermore, the user can define which matched parameters should be retained as part of the solution. All other parameters are re-matched by the graph transformation engine when the respective rule is executed again. Considering the *Stack* example, the user has probably an interest in preserving the stack from which the load is shifted (*fromId* parameter) and the stack to which the load is shifted (*toId* parameter).



Variables			Objectives	Constraints	Attributes
rule = shiftLeft fromId = 'Stack 2' toId = 'Stack 1' amount = 3	rule = shiftLeft fromId = 'Stack 4' toId = 'Stack 3' amount = 3	*Placeholder*	StdDev = 0.89443 Length = 2.0		Executions = [true, true, true] AggregatedFitness = 2.89443 CrowdingDistance = 1.01801 Rank = 0 ResultModel = 4 4 3 9 5

Fig. 3: One solution with two rule applications for the Stack example

Solution Repair. Even though constraints can be used to specify the validity of solutions, a solution that is a product of re-combining two other solutions might have rule applications that are not executable. By default, rules that can not be executed are ignored in Henshin. However, this behavior might not be satisfactory in some cases. Therefore we provide two repair strategies in our framework. The first one replaces all non-executable rule applications with rule application placeholders, preventing Henshin from executing them and removing the opportunity to use them again in another solution created through re-combination. The second strategy replaces each non-executable rule application with a random, executable rule application. This may have an impact on the overall solution quality as other rule applications might become non-executable and may also need to be replaced. Furthermore, the resulting solution may be a solution that is quite different from the original solution. Of course, depending on the chosen algorithm and the actual constraints of the solutions, a user can also select a dedicated re-combination operator that is able to consider some constraints, e.g., the partially matched crossover (PMX) [21] can preserve the order of variables within a solution.

3.4. Exploration Configuration

As mentioned previously, the proposed approach is algorithm-agnostic. Therefore additional algorithm-specific exploration options need to be configured by the user. Reusing MOEA, we can use the following evolutionary algorithms out of the box: NSGA-II, eNSGA-II, NSGA-III, eMOEA, and Random Search. Furthermore a set of selection and crossover operators are provided, which can also be reused. Additionally, we provide a base for local search algorithms from which we have implemented the Random Descent and Hill Climbing algorithms as proof-of-concept. A user may choose to develop further algorithms or integrate existing ones from the jMetal library[4], the PISA library[5] and the BORG MOEA Framework[6], for which adapters or specific plug-ins are already provided by MOEA. For illustration purposes, we discuss the options of evolutionary search configurations and local search configurations for the Stack example. Listing 3 depicts the common options of those two configurations.

Listing 3: Textual representation of generic search configurations for the Stack example

```
initialModel = 'model_five_stacks.xmi';             // as depicted in Figure 1b
rules = 'stack.henshin';                            // as depicted in Figure 2
rules.shiftLeft.amount = new RandomInteger(1, 5); // bounds for user parameter
rules.preserveParameters = [ rules.shiftLeft.fromId, rules.shiftLeft.toId ]; // matched parameters
solution.repairer = new PlaceholderSolutionRepairer();
```

Evolutionary Search Configuration. Evolutionary search algorithms are a subset of population-based search algorithms that deploy selection, crossover, and mutation operators to improve the fitness of the solutions in the population in each iteration (the first population is usually generated randomly). The *selection operators* can be defined generically and choose which solutions of the population should be considered for re-combination. An example for a selection operator would be deterministic tournament selection, which takes *n* random candidate solutions from the population and allows the best one to be considered for re-combination. The *crossover operator* is responsible for creating new solutions based on already existing ones, i.e., re-combining solutions into new ones. Presumably, traits which make the selected solutions fitter than other solutions will be inherited by the newly created solutions. In our case, each solution is represented as a *sequence* of rule applications for which many generic operators already exist, e.g., the one-point crossover operator which splits two solutions at a random point and merges them crosswise. The *mutation operators* are used to introduce slight, random changes into solution candidates. This guides the algorithm into areas of the search space that would not be reachable through recombination alone and avoids the convergence of the population towards a few elite solutions. To take the semantics of transformation rules into account, we have introduced three dedicated mutation operators. The first operator replaces random transformation rules by placeholders, reducing the actual solution length. The second operator applies a final transformation rule on the resulting model without changing the actual solution length, and the third operator varies the user parameters of a rule application based on the parameters bounds (cf. Section 3.3).

[4]http://jmetal.sourceforge.net
[5]http://www.tik.ee.ethz.ch/pisa/
[6]http://borgmoea.org/

Listing 4: Textual representation of an evolutionary search configuration for the Stack example

```
solution.nrVariables = 8;
fitness.objective['Standard Deviation'] = new OCLQueryDimension('...'); // as shown in Listing 2
fitness.objective['Solution Length'] = new SolutionLengthDimension();   // actual length of solution
search = new NSGAII();
search.populationSize = 50;
search.maxNrIteration = 100;
search.selection = [ new TournamentSelection(2) ];
search.crossover = [ new OnePointCrossOver(1.0) ]; // application probability: 100%
search.mutation  = [ new ParameterMutation(0.2), new PlaceholderMutation(0.15); ];
```

Local Search Configuration. Local search algorithms maintain one solution at a time and try to improve it in each iteration. Improvement depends on the *solution comparison method* the user selects, e.g., comparison based on objective, constraint or attribute values. The initial solution may be given by the user or can be generated randomly. In each iteration, the algorithm may take a step to a neighbor solution, i.e., a solution that is a slight variation of the current solution. The calculation of neighbors from the current solution can be done generically using a *neighborhood function*. How many neighbors are evaluated and whether only fitter neighbors are accepted as the next solution depends on the respective algorithm. In our framework, we provide two neighborhood functions. Following the principle of re-use, the first function uses one of the previously defined mutation operators to introduce slight changes into the current solution. Depending on the operator, this function may produce an infinite number of neighbors, e.g., when varying floating point rule parameter values. In such a case, an upper bound on the number of calculated neighbors can be specified. The second neighborhood function adds an additional, random rule application to the current solution, increasing its solution length. Here, a user may specify an upper bound on the solution length.

Listing 5: Textual representation of a local search configuration for the Stack example

```
fitness.objective['Standard Deviation'] = new OCLQueryDimension('...'); // as shown in Listing 2
search =  new HillClimbing();
search.maxNrEvaluations = 2000;
search.neighborhoodFunction = new IncreasingNeighborhoodFunction(100);  // maximum 100 neighbors
search.comparison = new ObjectiveComparison('Standard Deviation');
search.initialSolution = new Solution(0);                               // empty solution
```

3.5. Analysis and Statistics

Besides the orchestration of the transformation rules, the resulting models from those rules and their corresponding objective and constraint values, further analytical information about the search process and the results can be collected during the execution of an algorithm. This data can then be used to either influence the execution of the algorithm itself or to perform advanced analysis techniques after the execution has finished. In our framework, we provide ways to print detailed information about the on-going search process on the console and to terminate a single run of an algorithm depending on different criteria, e.g., after a certain time limit has been reached or a given solution has been found.

For further analysis after the algorithm(s) have executed, we can re-use different mechanisms provided by MOEA to compare two or more solution sets. Typically, one set is the so-called *reference set* which contains the known Pareto optimal solutions and to which the other solution sets are compared. If the Pareto optimal solutions are not known a priori, which is the case for many real-world problems, an approximation to this set may be generated by executing different algorithms multiple times. By measuring the distance of the found solution set to the reference set, we can calculate several indicators such as hypervolume, generational distance or the maximum Pareto front error. Pairing this distance with statistical tests performed by MOEA, we can evaluate the hypothesis that one algorithm is significantly better than another algorithm in a specific scenario. Currently supported statistical tests include among others the Kruskal-Wallis One-Way Analysis of Variance by Ranks test [22, 23] or the Mann-Whitney U test [23]. An example output of a statistical analysis for the hypervolume indicator based on the configuration shown in Listing 4 is shown in Table 1. We can see that all three algorithms perform equally well and that the statistical test indicates that they are interchangeable with regard to this indicator (*indifferent*). All collected and calculated data can also be used to plot graphs giving a better overview about the algorithm executions. Figure 4 depicts the

	NSGA-III	ε-MOEA	NSGA-II
Aggregate	0.314050	0.314050	0.314050
Min	0.219453	0.184262	0.219453
Max	0.314050	0.314050	0.314050
Median	0.314050	0.314050	0.314050
Values	[0.219453, 0.314050, . . .]	[0.184262 0.314050, . . .]	[0.219453 0.314050, . . .]
Count	50	50	50
Indifferent	ε-MOEA NSGA-II	NSGA-III NSGA-II	ε-MOEA NSGA-III

Table 1: Excerpt of Hypervolume statistics

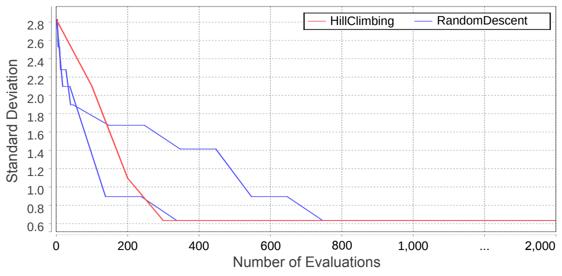


Fig. 4: Standard Deviation convergence for local search

convergence of the standard deviation fitness dimension over time for multiple local search execution runs as configured in Listing 5. Each line corresponds to one run of the respective algorithm.

3.6. Implementation

We have implemented our approach by providing a bridge between Henshin and the MOEA framework. It consists mainly of the encoding of the decision variables in the form of transformation rule applications. The solution, the random solution generation, the created generic operators, and the additional solution configurations rely solely on this encoding, independent of the actual model transformation engine. The complete code as well as the examples presented in this paper can be found on our project website[7].

4. Evaluation

In this section we present an evaluation of our approach based on three case studies. In particular, we are interested in answering the following two research questions (RQ).

RQ1. Applicability: Is our approach applicable to classic problems in model-based software engineering?

RQ2. Overhead: How much overhead is introduced by our approach compared to a native encoded solution?

Case Study Design and Requirements. To answer RQ1 we evaluate our approach in three different case studies that target different problem areas and differ in their level of complexity with regard to the rule size. To answer RQ2, we compare our approach with a native encoding for one of the case studies. For each case study we need the input in the format described in Section 3, i.e., an Ecore-based meta-model representing the problem domain and Henshin transformation rules to manipulate instances of that problem domain. All case study experiments need to be conducted under the same conditions, i.e., running on the same machine with equal search algorithms and operators. The case studies are explained in the next section.

4.1. Case Study Setup

This section explains the three case studies used to evaluate our approach and framework and to answer our RQs. The first case study is from the Stack problem domain and has been introduced throughout the paper. The second and third case studies represent classical problems of model-based software engineering.

Stacks. The problem domain is a set of stacks with different loads that are connected in a circular way. The two objectives that should be minimized are the *standard deviation of the loads* and the *solution length*.

[7]`https://code.google.com/p/momot/`

Modularization. This is a classic problem in software architecture design [24, 25, 26]. The goal of modularization is to group a number of classes that have inter-dependencies into modules or components to minimize coupling and maximize cohesion. *Coupling* refers to the dependencies among classes of different components and thus to the dependencies among the respective components. Typically, low coupling is preferred as this indicates that each component covers separate functionality aspects of the system, improving the maintainability and testability of the overall system [27]. On the contrary, the *cohesion* within a single component, i.e., the relations of classes within one component, should be maximized to ensure that no class that is not part of the components functionality is included in that component. A problem instance of this case study consists of a set of classes and their inter-dependencies. To manipulate this instance we need to (*i*) create new components and (*ii*) assign classes to an existing component. A valid solution for the modularization problem assigns each class to exactly one component and has no empty components.

Class Diagram Restructuring. The third case study is taken from the Transformation Tool Contest (TTC) of 2013 [28]. The aim of the TTC series is to compare the expressiveness, the usability, and the performance of graph transformation tools along a number of selected case studies. Specifically, we use the *Class Diagram Restructuring* case study [29, 30], which consists of an in-place refactoring transformation on UML class diagrams. A problem instance consists of a set of classes and their attributes as well as possible inheritance relationships between the classes. The goal is to remove duplicate attributes from the overall class diagram, and to identify new classes which abstract data features shared in a group of classes in order to minimize the number of entities, i.e., classes and attributes. The three ways used to achieve this objective are (*i*) *pulling up* common attributes of all direct subclasses into the super-class, (*ii*) *extracting a super-class* for duplicated attributes of classes that already have a super-class, and (*iii*) *creating a root class* for duplicated attributes of classes that have no super-class.

4.2. Measures

To assess the applicability of our approach (*RQ1*), we use all three case studies as they are known problems that have been extensively discussed in the literature. In particular, we consider our approach to be applicable for a specific case study if the respective problem domain can be represented using our approach. This will indicate whether the formalisms used in our approach, i.e., using meta-models and graph transformation rules, are expressive enough for real-world problems. Finally, to assess the overhead of our approach (*RQ2*), we compare the time it takes to obtain the solutions for a particular problem (total runtime performance) of both our approach and a native implementation in the MOEA framework. The overhead of our approach will be evaluated for the Stack domain by varying the population size parameter. To ensure the sanity of the solutions, we evaluate if the retrieved solutions violate any constraint or have contradicting objective values.

4.3. Results

This section describes the experiments that we have conducted with the three case studies. Based on the obtained results we discuss the answers to our research questions. All results are based on at least 20 runs of the NSGA-II algorithm, which we deem sufficient for our overhead analysis, and all shown numbers are the average of these executed runs. The artifacts created for this evaluation are available on our project website.

RQ1. To answer the first research question, we have modeled the problem domain of all case studies as Ecore meta-models and have developed rules that manipulate instances of these meta-models.

The applicability of the *Stack* example has been shown throughout the paper. Applying the approach on the input model shown in Figure 1b, we quickly find the shortest sequence of rule applications (three) that leaves five pieces in each stack as well as the other solutions of length two, one and zero of the Pareto-set.

Regarding the *Modularization* case study, we have quickly modeled both the problem domain as a meta-model with three classes and the transformation rules. Since both rules, i.e., creating components and assigning classes to components, rely solely on random generation (of component names) and pure graph pattern matching without any additional parameters, they can be directly implemented as transformation rules. The necessary objectives and constraints can be specified using simple Java methods. We have executed different problem instances and manually performed a successful sanity check.

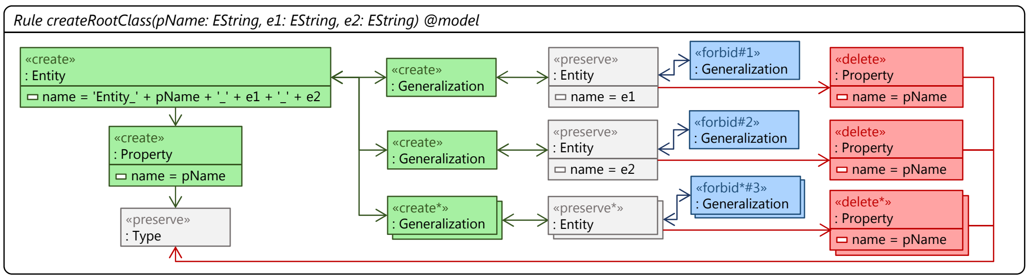


Fig. 5: Rule to create a root class for a common attribute *pName* with three NACs (*forbid*) and one nested rule ('∗')

The *Class Diagram Restructuring* case study is the most complex one with regard to rule complexity. While it is possible to translate all three manipulations into graph transformation rules, each rule needs at least one NAC and consists of at least one nested rule. The rule for creating root classes is shown in Figure 5. A nested rule in Henshin (indicated by a '∗' in the action name in Figure 5) is executed as often as possible if the outer rule matches. Therefore nested rules can lead to a large set of overall rule matches, making the application of such rules more expensive. The NACs have been implemented both directly as graph patterns (indicated as *forbid* action in Figure 5) and as OCL constraints. For example, the NAC of pulling up attributes shown in Listing 6 ensures that all sub-classes of the class with the name *eName* have an attribute with name *pName* before that attribute is pulled up. Both parameters, *eName* and *pName*, are matched by the graph transformation engine automatically.

Listing 6: OCL-NAC for pulling up attributes specified in the context of the class diagram

```
self.entitys->select(e | e.name = eName).specialization->collect(g | g.specific)
            ->forAll(e | e.ownedAttribute->exists(p | p.name = pName))
```

Furthermore, in this case study, the order in which the three rules are executed has a direct effect on the quality of the resulting model. In this sense, pulling up attributes (rule 1) should have priority over extracting super classes (rule 2) and this one over creating root classes (rule 3). Besides, for rules 2 and 3, the largest set of classes containing a common attribute should be chosen. Such a definition of rule priority is not directly possible in our approach as we choose an applicable rule at random. Therefore, this prioritization must be reflected in the specification of the objectives. We do so by putting an additional weight on the number of attributes when calculating the number of entities in the model, resulting in the OCL objective shown in Listing 7, referring to all entities and attributes in the model, respectively.

Listing 7: OCL objective for calculating the weighted number of classes (entity) and attributes (property)

```
properties->size() * 1.1 + entities->size()
```

In fact, in comparison with other approaches that provide solutions to this problem [30], defining such a prioritization in the objectives can prove to be more flexible as we can easily add new objectives or modify existing ones in order to obtain a different refactoring, while other approaches need to change the rules directly which requires a deeper understanding of the inter-dependencies between the rules.

RQ2. To evaluate the overhead of our approach, we compare the runtime performance of our approach with the performance of a native implementation in the MOEA framework for the Stack case study. The native implementation only uses classes from the MOEA framework and encodes the Stack problem as a sequence of binary variables or bit sets, respectively. While the Integer value of the binary variable indicates how much load is shifted, the position of the binary variable in the sequence indicates from which stack the load is shifted. Therefore we have as many binary variables as there are stacks in the model. In addition, one bit in each binary variable is added as a discriminator for indicating whether the load is shifted to the left or to the right. The number of bits to represent the load value is based on the highest load in the provided stack instance, e.g., to represent 15 we need 4 bits (`1111`) and to represent 16 we need 5 bits (`10000`).

To compare the two approaches, both are run with the same algorithm, NSGA-II, and with the same number of evaluations. The number of variables is set to the number of stacks. However, due to the differences in the encoding, we can not use the exact same mutation operators. In our approach we vary the *load* parameter and introduce placeholders as explained in Section 3. In the native approach, we apply a bit-flip mutation with the same probability as our parameter mutation and apply a mutation operator that sets one of the variables to zero with the same probability as the placeholder mutation. We let the load parameter in our approach vary between zero and the highest number that can be represented in the native solution. To execute the experiment, we fix the problem complexity with 100 stacks having a load between 0 and 200 and stop after 10000 evaluations. To gain insight into the performance, we vary the population size and the number of iterations respectively, the sum always being 10000 evaluations, so that the number of evaluations equals the number of iterations times the population size. Each execution consists of 20 algorithm runs.

Figure 6 depicts the results retrieved from these executions. The solid lines represent the average runtime of all runs while the vertical lines through each point indicate the minimum and the maximum runtime encountered. From the experiments we can observe that the native encoding has a stable runtime performance between one and three seconds while the runtime performance of our approach has more variation in each execution and grows linearly with the population size. Furthermore, our approach performs slower in all execution scenarios. This observed loss in performance can be explained two-fold. First, there is a slight difference in the performance of the applied mutation operators. While the bit-flip operator is really fast, the creation of a random parameter is slightly more expensive due to cost associated with creating random values. Second and more importantly, by using graph transformation rules instead of more dedicated

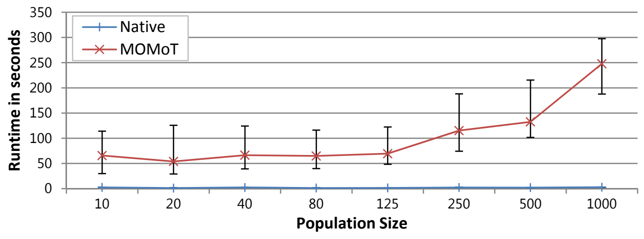


Fig. 6: Comparison of total runtime: Native encoding vs MOMoT

Iterations	Iteration Average		Population Creation	
1000	66.64ms	0.10%	678ms	1.03%
500	107.98ms	0.20%	1002ms	1.86%
250	265.45ms	0.40%	2185ms	3.29%
125	518.85ms	0.80%	3991ms	6.15%
80	868.07ms	1.25%	5374ms	7.74%
40	2881.34ms	2.50%	17253ms	14.97%
20	6628.76ms	5.00%	24276ms	18.31%
10	24791.99ms	10.0%	75974ms	30.64%

Table 2: Population Creation Overhead

encodings, we inherit the complexity of graph pattern matching and match application, which are expensive tasks on their own. This is especially evident when creating a large initial population in the first iteration, where we have to find and apply a lot of random matches, which takes a large proportion of the overall execution time. Table 2 depicts the runtime overhead introduced by the population creation compared to the expected average runtime of an iteration. The calculation of random matches for each solution is also a source of the variation of the overall runtime as it depends on which matches have been generated before. Subsequent iterations of the algorithm where we create new solutions through re-combination and thus only have to apply matches run much faster. So although Henshin uses constraint solving to tackle the problem of finding pattern matches more quickly, a certain loss in performance can not be avoided.

Our approach is consequently applicable for model-based software engineering problems. However, due to the complexity of graph pattern matching, there is a clear trade-off between the expressiveness provided by MDE through meta-modeling and model transformations and the performance of a more dedicated encoding. Inventing such a dedicated encoding is a creative process and is in most cases not as straight forward as in the Stack case study. Furthermore, once a dedicated encoding has been defined, integrating changes may become quite expensive. For instance, introducing a new manipulation that can reduce the load of a stack by half through compression in the native encoding requires at least the addition of another discriminator bit and the adaptation of the encoding interpretation done in the fitness function. By contrast, in our approach, one only needs to implement the manipulation as a model transformation rule. The complexity of finding a good, dedicated encoding becomes even more evident when many diverse manipulations with a varying number of parameters of different data types need to be represented in the solution. Additionally, a dedicated encoding may make assumptions about the deployed search algorithm, hampering the switch between different algorithms. Both of these drawbacks are mitigated in our approach, where the parameters and their data types are part of a transformation rule and where the switch to a different algorithm can be done by changing one line in the search configuration. Indeed, this ease of use is especially important for modelers, who are familiar with MDE technologies such as meta-modeling, model transformations, and OCL, but may find it challenging to express their problem with a dedicated encoding, corresponding operators and a fitness function as well as converting the obtained solutions back onto model level. The use of dedicated encodings is further complicated by the fact that there is often no agreed upon solution for solving a specific problem. For instance, whereas the works in [25, 26] both address the problem of refactoring at design level using a modeling approach, each of them proposes a different encoding.

Summing up, we conclude that while our approach is applicable for many problems, it is currently not suited for time-critical problems due to the overhead introduced by the generic encoding based on model transformations. Although such an encoding is useful for modelers, the cost-effectiveness of our encoding needs to be analyzed further.

4.4. Threats to Validity

In this subsection, we elaborate on several factors that may jeopardize the validity of our results.

Internal validity — Are there factors which might affect the results of this case study? First, a prerequisite of our approach is for the user to be able to create class diagrams (meta-models) and define rule-based systems. While this task was simple for us as modelers, people from other domains may find these tasks more challenging. Second, although we have sanity-checked all solutions, we only had a reference set of optimal solutions for one example (the modularization case study). Further investigation may be needed to ensure that our approach does not introduce problems hindering the algorithm from finding the optimal solutions.

External validity — To what extent is it possible to generalize the findings? Even though we have selected three case studies from different areas with varying degrees of complexity, the number of case studies may still not be representative enough to argue that our approach can be applied on any model-based software engineering problem. Therefore, additional case studies need to be conducted to mitigate this threat. Furthermore, we have used Henshin as model transformation language to express in-place model transformations. This means that additional studies are needed in order to know how integrable other model transformation languages are in our approach and to consider out-place model transformations. As part of our future work, we plan to investigate these issues and try to define a minimal set of requirements on the kinds of notations and transformation languages that are amenable to be directly addressed by our approach.

5. Related Work

With respect to the contribution of this paper, we discuss three main threads of related work. First, 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 done in the related field of program transformation.

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) [31, 32, 33] 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 [34, 35, 36, 37, 38, 39]. 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.

Searching for transformation rule applications with search-based optimization techniques for high-level change detection has been presented in [40]. 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 give 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 [41], 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. The authors in [42] 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. Another work reported in [43] also addresses the problem of finding optimal sequences of rule applications, but they deal with population-based search techniques. Thereby, this work is considered as a multi-objective exploration of graph transformation systems, where they apply NSGA-II [13] to drive rule-based design space exploration. Our presented work has 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 use the most appropriate one for a given transformation problem. Finally, the authors in [44] 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.

Program transformation is a field closely related to model transformation [45], thus, similar problems are occurring in both fields. One 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 [46, 47]. 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. However, we consider the instantiation of our framework for the problem of program transformation in combination with model-driven reverse engineering tools [3] as an interesting subject for future work to further evaluate our approach.

6. Conclusion and Future Work

In MDE, modelers represent problem domains by means of meta-models and model transformations are used to manipulate their instances, i.e, models. Most model transformation approaches are rule-based and require a form of rule orchestration, i.e., the specification of the rules ordering and the specification of rule parameters if necessary. Depending on the number of rules, the number of parameters, and the objectives that should be achieved by the transformation, finding a desired rule orchestration is a complex task. Therefore, in this paper we have proposed to apply SBSE techniques to support the modeler in this task. More specifically, we have presented an algorithm-agnostic approach that is able to support various optimization techniques while remaining in the model engineering technical space. Furthermore, we have shown that our approach produces correct and sound solutions w.r.t. the constraints and the objectives, and that it can be applied for typical model-based software engineering problems. By staying in the model engineering technical space and using the transformation rules directly in the encoding for the search process, we produce solutions that can be easily understood by modelers. However, when we consider models as graphs and transformation rules as graph patterns that must be matched, the problem of finding and applying matches is computationally expensive and reduces the runtime performance. Here we could clearly identify a trade-off of expressiveness and runtime performance in the evaluation of our approach in comparison with traditional encodings.

Although first results seem promising, our approach faces some limitations which we plan to address in the future. First of all, we will further investigate the limitations of our approach with respect to runtime performance. In particular, we foresee to work on speeding-up the graph pattern matching and the interpretation of OCL statements and we will investigate other performance measures such as the memory consumption. Second, we have assumed that by staying in the model engineering technical space, modelers can directly use our approach transparently. However, we want to explore more empirical case studies to evaluate the usability of our approach and the cost-effectiveness of using a generic, more expressive but less performant encoding. Third, we plan to provide mechanisms to guide the users of our framework in selecting appropriate search configuration options. Suggestions might be made for example based on the number of objectives, the number of constraints, and whether the ordering of the rules is important or not. Finally, we would like to study the applicability of our approach in larger contexts, such as the proper selection of cloud patterns when moving a legacy application to the cloud in order to improve its non-functional properties [48] and to support approximate model transformations [49], i.e., trading precision for performance.

Acknowledgements

The authors thank the reviewers for their insightful and constructive comments and suggestions. This work is funded by the European Commission under ICT Policy Support Programme, grant no. 317859.

References

[1] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers, 2012.

[2] S. Sendall, W. Kozaczynski, Model Transformation: The Heart and Soul of Model-Driven Software Development, IEEE Software 20 (5) (2003) 42–45.

[3] H. Bruneliere, J. Cabot, F. Jouault, F. Madiot, MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering, in: Proc. of ASE, IEEE/ACM, 2010, pp. 173–174.

[4] K. Czarnecki, S. Helsen, Feature-based Survey of Model Transformation Approaches, IBM Systems Journal 45 (3) (2006) 621–645.

[5] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. Selim, E. Syriani, M. Wimmer, Model Transformation Intents and Their Properties, Software and System Modeling (2014) 1–38.

[6] I. Kurtev, State of the Art of QVT: A Model Transformation Language Standard, in: Proc. of AGTIVE, Vol. 5088 of LNCS, Springer, 2008, pp. 377–393.

[7] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A model transformation tool, Science of Computer Programming 72 (1–2) (2008) 31 – 39.

[8] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, D. Varró, VIATRA - visual automated transformations for formal verification and validation of UML models, in: Proc. of ASE, IEEE/ACM, 2002, pp. 267–270.

[9] M. Kessentini, P. Langer, M. Wimmer, Searching models, modeling search: On the synergies of SBSE and MDE, in: Proc. of CMSBSE @ ICSE, IEEE, 2013, pp. 51–54.

[10] M. Harman, The Current State and Future of Search Based Software Engineering, in: Proc. of FOSE @ ICSE, 2007, pp. 342–357.

[11] I. Kurtev, M. Aksit, J. Bézivin, Technological Spaces: An Initial Appraisal, in: Proc. of CoopIS, DOA, and ODBASE, 2002.

[12] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer, Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations, in: Proc. of MODELS, Vol. 6394 of LNCS, Springer, 2010, pp. 121–135.

[13] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II, IEEE Transactions on Evolutionary Computation 6 (2) (2002) 182–197.

[14] K. Deb, H. Jain, An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints, IEEE Trans. on Evolutionary Computation 18 (4) (2014) 577–601.

[15] K. Deb, M. Mohan, S. Mishra, A Fast Multi-objective Evolutionary Algorithm for Finding Well-Spread Pareto-Optimal Solutions, Tech. Rep. 2003002, Indian Institute of Technology Kanpur (2003).

[16] G. Taentzer, AGG: A Graph Transformation Environment for Modeling and Validation of Software, in: Proc. of AGTIVE, Vol. 3062 of LNCS, Springer, 2004, pp. 446–453.

[17] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude – A High-Performance Logical Framework, Vol. 4350 of LNCS, Springer, 2007.

[18] J. de Lara, H. Vangheluwe, AToM3: A Tool for Multi-formalism and Meta-modelling, in: Proc. of FASE, Vol. 2306 of LNCS, Springer, 2002, pp. 174–188.

[19] J. E. Rivera, F. Duran, A. Vallecillo, A Graphical Approach for Modeling Time-dependent Behavior of DSLs, in: Proc. of VL/HCC, IEEE, 2009, pp. 51–55.

[20] E. Biermann, C. Ermel, G. Taentzer, Lifting Parallel Graph Transformation Concepts of Model Transformation based on the Eclipse Modeling Framework, Electronic Communications of the EASST 26 (2010) 1–19.

[21] D. E. Goldberg, R. Lingle Jr., Alleles, loci, and the traveling salesman problem, in: Proc. of ICGA, Lawrence Erlbaum Associates, 1985, pp. 154–159.

[22] W. H. Kruskal, W. A. Wallis, Use of Ranks in One-Criterion Variance Analysis, Journal of the American Statistical Association 47 (260) (1952) 583–621.

[23] D. J. Sheskin, Handbook of Parametric and Nonparametric Statistical Procedures, 3rd Edition, Chapman & Hall/CRC, 2003.

[24] K. Praditwong, M. Harman, X. Yao, Software Module Clustering as a Multi-Objective Search Problem, IEEE Transactions on Software Engineering 37 (2) (2011) 264–282.

[25] O. Seng, J. Stammel, D. Burkhart, Search-based Determination of Refactorings for Improving the Class Structure of Object-oriented Systems, in: Proc. of GECCO, ACM, 2006, pp. 1909–1916.

[26] M. Harman, L. Tratt, Pareto Optimal Search Based Refactoring at the Design Level, in: Proc. of GECCO, ACM, 2007, pp. 1106–1113.

[27] E. Yourdon, L. L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, 1st Edition, Prentice-Hall, Inc., 1979.

[28] P. V. Gorp, L. M. Rose, C. Krause (Eds.), Proc. of the 6th Transformation Tool Contest (TTC'13), Vol. 135 of EPTCS, Open Publishing Association, 2013.

[29] K. Lano, S. Kolahdouz Rahimi, Case study: Class diagram restructuring, in: Proc. of TTC, Vol. 135 of EPTCS, Open Publishing Association, 2013, pp. 8–15.

[30] S. Kolahdouz-Rahimi, K. Lano, S. Pillay, J. Troya, P. V. Gorp, Evaluation of model transformation approaches for model refactoring, Science of Computer Programming 85, Part A (0) (2014) 5 – 40.

[31] D. Varró, Model Transformation by Example, in: Proc. of MODELS, Vol. 4199 of LNCS, Springer, 2006, pp. 410–424.

[32] M. Wimmer, M. Strommer, H. Kargl, G. Kramler, Towards Model Transformation Generation By-Example, in: Proc. of HICSS, IEEE, 2007, p. 285b.

[33] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, M. Wimmer, Model Transformation By-Example: A Survey of the First Wave, in: Conceptual Modelling and Its Theoretical Foundations, Vol. 7260 of LNCS, Springer, 2012, pp. 197–215.

[34] M. Kessentini, H. A. Sahraoui, M. Boukadoum, Model Transformation as an Optimization Problem, in: Proc. of MODELS, Vol. 5301 of LNCS, Springer, 2008, pp. 159–173.

[35] M. Kessentini, A. Bouchoucha, H. A. Sahraoui, M. Boukadoum, Example-Based Sequence Diagrams to Colored Petri Nets Transformation Using Heuristic Search, in: Proc. of ECMFA, Vol. 6138 of LNCS, Springer, 2010, pp. 156–172.

[36] M. Kessentini, H. A. Sahraoui, M. Boukadoum, O. Benomar, Search-based model transformation by example, Software and System Modeling 11 (2) (2012) 209–226.

[37] I. Baki, H. A. Sahraoui, Q. Cobbaert, P. Masson, M. Faunes, Learning Implicit and Explicit Control in Model Transformations by Example, in: Proc. of MODELS, Vol. 8767 of LNCS, Springer, 2014, pp. 636–652.

[38] M. Faunes, H. A. Sahraoui, M. Boukadoum, Genetic-Programming Approach to Learn Model Transformation Rules from Examples, in: Proc. of ICMT, Vol. 7909 of LNCS, Springer, 2013, pp. 17–32.

[39] H. Saada, M. Huchard, C. Nebut, H. A. Sahraoui, Recovering model transformation traces using multi-objective optimization, in: Proc. of ASE, IEEE/ACM, 2013, pp. 688–693.

[40] A. ben Fadhel, M. Kessentini, P. Langer, M. Wimmer, Search-based detection of high-level model changes, in: Proc. of ICSM, IEEE, 2012, pp. 212–221.

[41] D. Efstathiou, J. R. Williams, S. Zschaler, Crepe complete: Multi-objective optimisation for your models, in: Proc. of CMSEBA @ MODELS, 2014.

[42] J. Denil, M. Jukss, C. Verbrugge, H. Vangheluwe, Search-Based Model Optimization Using Model Transformations, in: Proc. of SAM, Vol. 8769 of LNCS, Springer, 2014, pp. 80–95.

[43] H. Abdeen, D. Varró, H. A. Sahraoui, A. S. Nagy, C. Debreceni, Á. Hegedüs, Á. Horváth, Multi-objective optimization in rule-based design space exploration, in: Proc. of ASE, IEEE/ACM, 2014, pp. 289–300.

[44] J. Shelburg, M. Kessentini, D. Tauritz, Regression Testing for Model Transformations: A Multi-objective Approach, in: Proc. of SSBSE, Vol. 8084 of LNCS, Springer, 2013, pp. 209–223.

[45] E. Visser, A survey of rewriting strategies in program transformation systems, ENTCS 57 (2) (2001) 109–143.

[46] D. Fatiregun, M. Harman, R. M. Hierons, Search Based Transformations, in: Proc. of GECCO, Vol. 2724 of LNCS, Springer, 2003, pp. 2511–2512.

[47] D. Fatiregun, M. Harman, R. M. Hierons, Evolving transformation sequences using genetic algorithms, in: Proc. of SCAM, IEEE, 2004, pp. 66–75.

[48] M. Fleck, J. Troya, P. Langer, M. Wimmer, Towards Pattern-Based Optimization of Cloud Applications, in: Proc. of CloudMDE @ MoDELS, Vol. 1242 of CEUR Workshop Proceedings, CEUR-WS.org, 2014, pp. 16–25.

[49] J. Troya, M. Wimmer, L. Burgueño, A. Vallecillo, Towards Approximate Model Transformations, in: Proc. of AMT @ MoDELS, Vol. 1277 of CEUR Workshop Proceedings, CEUR-WS.org, 2014, pp. 44–53.