Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2016; **28**:1081–1117
Published online 11 October 2016 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1804

Search-based model transformations

Martin Fleck[1,*,†], Javier Troya[2] and Manuel Wimmer[1]

[1]*Business Informatics Group, Institute of Software Technology and Interactive Systems, TUWien, Austria*
[2]*Department of Computer Languages and Systems, University of Seville, Seville, Spain*

ABSTRACT

Model transformations are an important cornerstone of model-driven engineering, a discipline which facilitates the abstraction of relevant information of a system as models. The success of the final system mainly depends on the optimization of these models through model transformations. Currently, the application of transformations is realized either by following the apply-as-long-as-possible strategy or by the provision of explicit rule orchestrations. This implies two main limitations. First, the optimization objectives are implicitly hidden in the transformation rules and their orchestration. Second, manually finding the best orchestration for a particular scenario is a major challenge due to the high number of possible combinations.

To overcome these limitations, we present a novel framework that builds on the non-intrusive integration of optimization and model transformation technologies. In particular, we formulate the transformation orchestration task as an optimization problem, which allows for the efficient exploration of the transformation space and explication of the transformation objectives. Our generic framework provides several search algorithms and guides the user in providing a proper search configuration. We present different instantiations of our framework to demonstrate its feasibility, applicability, and benefits using several case studies. Copyright © 2016 John Wiley & Sons, Ltd.

Received 21 September 2015; Revised 19 June 2016; Accepted 5 July 2016

KEY WORDS: model transformation; transformation orchestration; model-driven engineering; searchbased; software engineering

1. INTRODUCTION

Transformations are an important concept in computer science in general and in software engineering in particular, because, 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. In fact, when constructing a system, the design can be expressed in terms of models defined at different levels of abstraction that can be simulated [3]. Indeed, it is recommended to test and optimize the system already in the design phase [4], because errors that are detected in this phase are cheaper and easier to correct than those detected later on. Model transformations provide the essential mechanisms for manipulating models. For instance, they are crucial for refining the models in the design phase. Another well-known example of the application of model transformations is model refactoring, that is, improving the structure of models while preserving their observable behavior. Transformations can also be used to abstract software models such as UML class diagrams from existing source code using reverse engineering techniques [5], or to obtain some knowledge from the system, such as slicing the domain model out of a UML class

*Correspondence to: Business Informatics Group, Institute of Software Technology and Interactive Systems, TU Wien, Austria.
†E-mail: fleck@big.tuwien.ac.at

diagram [6]. Several distinct categories of model transformation languages have been identified [7, 8]. 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 their orchestration. A transformation orchestration includes the rule scheduling, that is, the definition of the order in which the rules are executed, and the rule parametrization, that is, the specification of values for the parameters of the rules. This orchestration can be defined implicitly or explicitly [7]. 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 [9] or Triple-Graph-Grammars [10], and many graph transformation languages [11–13]. Other languages offer mechanisms to explicitly define the rule scheduling. For instance, ATL [14] is a hybrid transformation language that offers the possibility of partially orchestrating the transformation by explicitly making calls to so-called lazy rules. Other languages provide more dedicated mechanisms to schedule rules, such as VIATRA [15], which offers rule scheduling using abstract state machines, and Henshin [16], which offers dedicated units for this purpose.

When developing a model transformation, the rules that manipulate the input models have to be defined. However, reasoning about how these rules can be applied in order to retrieve a model with certain characteristics is a non-trivial task suffering from three major drawbacks. First, the effect of a rule application on the characteristics of the resulting model is implicitly hidden in the behavior encoded by the rule. Second, if several rule applications to the same model exist, they may be in conflict or rule applications may enable each other [17], that is, a given rule may require some information produced by another rule. Third, the number of rule combinations may be very large or even infinite, especially when considering the input parameters a rule may have, making an exhaustive exploration of transformation orchestrations very difficult.

The idea to tackle this problem using search-based techniques is reflected in the recent approaches by Abdeen et al. [18] who integrate multi-objective optimization techniques to drive a rule-based design exploration, and Denil et al. [19] who integrate 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 previous work [20] on combining Search-Based Software Engineering (SBSE) [21] and MDE, we have introduced an algorithm-agnostic approach called *MOMoT* (Marrying Optimization and Model Transformations) [22] which formulates the problem of finding the best transformation orchestration for a given set of transformation rules as an optimization problem. Our approach loosely couples the MDE and SBSE worlds to allow the users to benefit from search-based techniques while staying in the model engineering technical space [23], that is, the search configuration and the computed solutions are provided at model level. In particular, we are focusing on how different SBSE techniques can be used to solve an optimization problem, on how we can support the users in configuring these techniques, and on the separation of the objectives of the transformation from the transformation itself. This separation enables us to reuse the same set of transformation rules for several problem scenarios.

Our approach is implemented within the Eclipse Modeling Framework (EMF)[1] and builds upon Henshin[2] [16] 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 [24], NSGA-III [25], and ε-MOEA [26], 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.

This article is an extension of a paper presented at the NasBASE 2015 conference [22]. Besides several incremental advancements, this article extends the previously introduced approach with the following aspects:

[1]Eclipse Modeling Framework: http://www.eclipse.org/modeling
[2]Henshin Project: http://www.eclipse.org/henshin
[3]MOEA Framework, version 2.8, available from http://www.moeaframework.org

Control Flow Units	The approach supports the orchestration of control flow units such as loops and conditions as part of the search. These control flow units are treated as a single step in the resulting transformation orchestration. Previously, the orchestration only supported transformation rules as steps. The use of these control flow units is investigated in an additional case study (cf. Section 4).
Configuration Language	We provide a configuration language that enables the user to configure necessary information to use our approach in a single model. Furthermore, when using this language, we provide content-assist to the user and collect information about the configuration. Using this information, we prevent invalid configurations from executing, warn the user about uncommon settings and inform the user about the different aspects of the configuration. This is especially useful for model engineers who are non-experts in SBSE techniques (cf. Section 3).
Extended Evaluation	Finally, we investigate the contributions of our approach by arguing about the benefits and drawbacks over a pure MDE approach. In this sense, we compare the configuration steps that an engineer needs to perform with our approach for solving a specific problem, provided the system is represented in the MDE domain (applying meta-modeling and model transformations), with how the problem is solved with current existing MDE approaches (cf. Section 4). Furthermore, we show that our approach is applicable to complex problems by applying it on a real-world system in one of the case studies.

The remainder of this paper is structured as follows. In Section 2, we give a short introduction to MDE, that is, meta-modeling, models and model transformation, and present a running example. Section 3 introduces our approach and describes the features of the framework that has been developed, in particular, which techniques we provide and how the framework can be used and 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 AND RUNNING EXAMPLE

This section introduces the background of this work, namely the basic notions of SBSE, models and meta-models as well as how to manipulate models with model transformations. Furthermore, we present a particular example to demonstrate these basic notions and that serves as the running example of this paper.

2.1. Search-based software engineering

Search-based software engineering [21] is a field that applies search-based optimization techniques to software engineering problems. Search-based optimization techniques can be categorized as metaheuristic 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 infeasible to use exact or enumerative approaches [27].

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 $f\colon X \mapsto Z$ maps a given solution to an objective vector in the objective space Z. A metaheuristic 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.

Metaheuristic approaches dealing with large or infinite search spaces are often divided in two groups, namely local search methods and evolutionary algorithms. The aim of local search methods is to improve one single solution at a time while evolutionary algorithms [28] manage a set of solutions, called a population, at once. Examples for local search algorithms are Tabu Search [29] or Simulated Annealing [30]. Examples for evolutionary algorithms include NSGA-II [24] and NSGA-III [25].

The number of objectives that need to be optimized, that is, 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. For multi-objective problems, the fitness function can also be denoted as a set of individual fitness functions $f(x)=[f_1(x),f_2(x),\ldots,f_m(x)]$ where m is the number of objectives. In order to have a uniform optimization direction for all functions, a maximization objective can be easily turned into a minimization objective by taking its negative value and vice versa. 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 [25]. A many-objective problem, as opposed to a multi-objective problem, is a problem with at least four objectives.

The type of the optimization problem also defines how different solutions can be compared in order to find good solutions to a problem. In the mono-objective case, a solution $x_1 \in X$ can be considered better than another solution $x_2 \in X$, if $f(x_1) < f(x_2)$ when the aim is to minimize the objective, or if $f(x_1) > f(x_2)$ when the aim is to maximize the objective. In the multi-objective and many-objective cases, the comparison is not that simple. One way to compare two objective vectors is to aggregate all objective values of each vector and compare the aggregated values. However, this is only possible if all values are on the same scale. Alternatively, we can use the notion of Pareto optimality to define whether one solution can be considered better than another. Under Pareto optimality, as defined in Equation 1 and in Equation 2 for strict inequality [21], one solution x_1 dominates another solution x_2 if it is better according to at least one of the objective values $f_i(x_1)$ and not worse according to all the others.

$$f(x_1) \geq f(x_2) \Leftrightarrow \forall i f_i(x_1) \geq f_i(x_2) \tag{1}$$

$$f(x_1) > f(x_2) \Leftrightarrow \forall i f_i(x_1) \geq f_i(x_2) \wedge \exists i f_i(x_1) > f_i(x_2) \tag{2}$$

Each non-dominated solution can be considered as an optimal trade-off between all objectives. However, there may exist several optimal trade-offs between the individual objectives. Therefore, an algorithm may yield a set of optimal solutions, often denoted as the Pareto-optimal set.

2.2. *Models and meta-models*

Meta-models are the means in MDE to specify the abstract syntax of modeling languages [31]. For defining meta-models, there are meta-modeling standards such as the Meta Object Facility (MOF) [32] which are mostly based on a core subset of the UML class diagrams, that is, classes, attributes, and references. A meta-model gives the intentional description of all possible models within a given language. Practically, meta-models are instantiated to produce models which are in essence object graphs, that is, they consist of objects (instances of classes) representing the modeling elements, object slots for storing values (instances of attributes), and links between the objects (instances of references), which have to conform to the UML class diagram describing the meta-model. Therefore, models are often represented in terms of UML object diagrams. A model has to conform to its meta-model which is often indicated by the *conformsTo* relationship (cf. Figure 1). In addition to the constraints defined by the meta-model, additional constraints may be defined based on the meta-model elements using a *constraint language*. The Object Constraint Language (OCL) [33] is a standardized and formal language to describe expressions, constraints, and queries on models. As

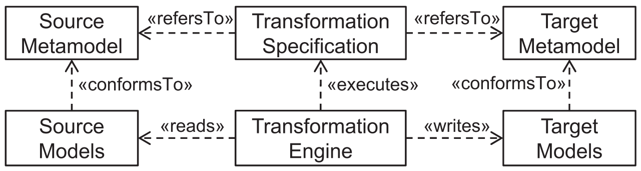


Figure 1. Model transformation pattern [7].

such, OCL is the language of choice for defining constraints going beyond simple multiplicity and type constraints defined by UML class diagrams and meta-models.

2.3. *Model transformations*

In a general sense, a model transformation is a program executed by a transformation engine which takes one or more models as input to produce one or more models as output as is illustrated by the model transformation pattern[4] [7] in Figure 1. In MDE, model transformations are used to solve different tasks [7, 34]. One important aspect is that model transformations are developed on the meta-model level, and thus, are reusable for all valid model instances.

In the MDE field, various model transformation kinds emerged in the last decade [7, 34]. A model transformation can be categorized as *out-place* if it creates new models from scratch, for example, reverse engineering code as models, or as *in-place* if it rewrites the input models until the output models are obtained, for example, as it is the case in model refactoring. In this paper, we focus on in-place model-to-model transformations, which can be expressed using graph transformation rules. In the software engineering process, these transformations are tailored at improving and refining the models in the design phase in order to validate them before the implementation starts [15, 35]. The applicability of graph transformations for model transformations builds upon the fact that most models exhibit a graph-based structure, for example, 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, that is, the output model. In general, a graph transformation rule $r=(L,R)$ consists of left-hand side (L) and right-hand side (R), which describe the model patterns and conditions to be matched (preconditions) and the effect of the changes to be applied (postconditions), respectively. The left-hand side of a rule may 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. To apply a rule, a graph transformation engine finds a match $m:L\mapsto M$ in the model M, removes elements from M which have an image in $L\setminus R$ and creates all elements of $R\setminus L$ in M. The result is a new graph on which further rules may be applied. In order to pass contextual information between rules, rules can also have parameters. There is a plethora of frameworks and languages to define these kinds of transformations, such as AGG [12], AToM^3 [36], AToMPM [37], e-Motions [11], Henshin [16], Maude [38], and VIATRA [15]. Because of the different possibilities of orchestrating transformations, a mechanism that automatically determines the optimal execution order and parametrization of rules for a specific scenario may ease the work of the transformation engineer substantially.

2.4. *Running example: modularization*

A well-known problem in software architecture design that can often not be solved through exhaustive approaches is the modularization problem [39–41]. The goal is to create high-quality object-oriented models by grouping classes into modules. Solving this problem implies to decide where classes belong and how objects should interact. Producing a class diagram where the right number of modules is chosen and a proper assignment of classes is realized is a non-trivial task, often requiring human judgment and decision-making [42]. In fact, the problem has an exponentially growing search space of potential class partitions given by the Bell number $B_{n+1}=\sum_{k=0}^{n}\binom{n}{k}B_k$. The nth

[4]Please note that the terms source/target models and input/output models are used synonymously.

of these numbers, B_n, counts the number of different ways a given set of n classes can be divided into modules. If there are no classes given (B_0), we can in theory produce exactly one partition (the empty set, $\emptyset$). The order of the modules as well as the order of the classes within a module does not need to be considered as the semantics of a class diagram does not depend on that order. Already starting from a low number of classes, the number of possible partitions is unsuitable for exhaustive search. For instance, with 15 classes, which are not many in a real-world example, the problem yields $1,382,958,545$ possible ways to create modules. Even when dismissing special cases where we group all classes into one module or every class into a separate module, the search space is still enormous.

Meta-modeling. In MDE, the first step to solve such a problem is to model the problem domain in terms of a meta-model. UML already considers this problem as part of its class diagram structure. However, for simplicity, we will work on a smaller version of such meta-model, as depicted in Figure 2. In our modularization meta-model, a system consists of modules and classes. Classes may depend on an arbitrary number of other classes (*dependsOn-providesFor* relationship), that is, if class A depends on class B, B is a provider for A. Modules are used to group classes into meaningful clusters. As we can see, each class may be in one module, while modules may contain many classes. An instance of such modularization meta-model is depicted in Figure 3. In the figure, the relationships between classes are shown in different colors to foster the figure's readability. It is a representation of *mtunis* [43], an operating system for educational purposes written in the Turing Language, which contains 20 classes and 57 dependencies among them. This means that there are, according to the Bell number described earlier, $51,724,158,235,372$ possibilities for grouping these classes into modules. Please note that the figure depicts the abstract model syntax, that is, the graph-based representation of the model information modulo its notation.

Model Transformations. To manipulate model instances in order to group classes into modules, we propose two rules depicted in Figure 4. In this figure, we use the Henshin notation. Since at the beginning, there are no modules in the input model (it only contains classes and the dependencies among them), we need a rule to create a module (*createModule*). Elements marked as *forbid* are part

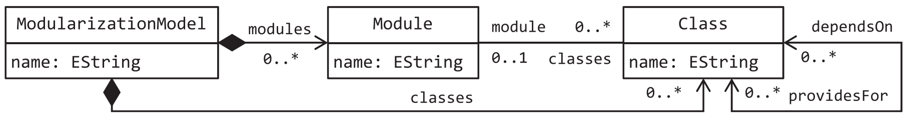


Figure 2. Modularization meta-model.

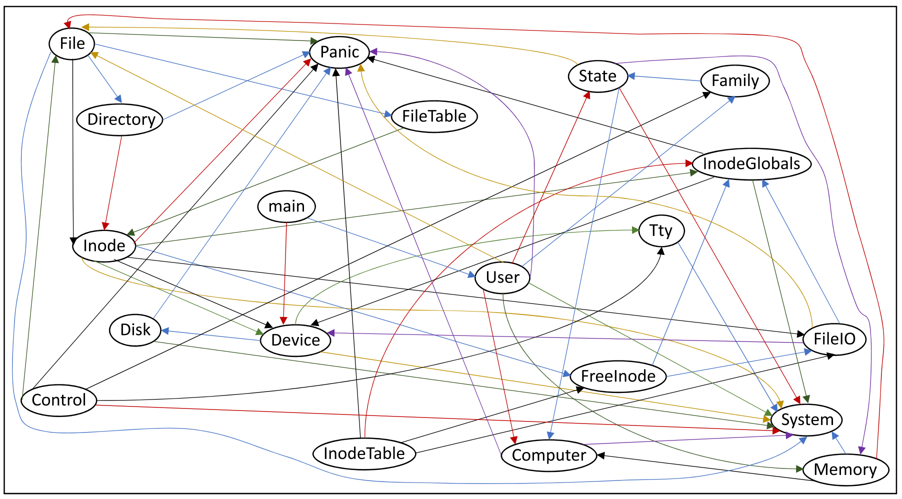


Figure 3. Modularization model instance (*mtunis* system).

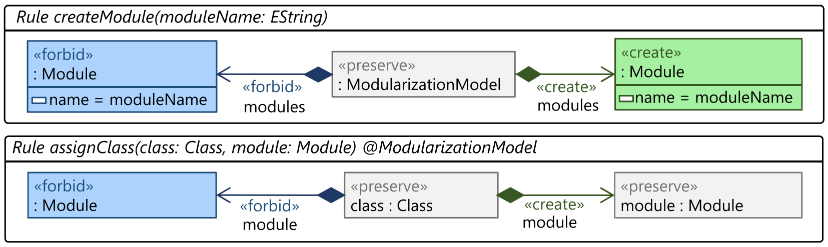


Figure 4. Rules to manipulate a modularization model.

of the rules negative application condition. This rule creates a module within the ModularizationModel with the provided name, only if a module with such a name does not already exist. Rule *assignClass* then enables the system to group a previously unassigned class into a module. All rules have parameters, namely *moduleName*, *module*, and *class*. While producing a match for these rules, all these input parameters acquire a value, that is, they are instantiated, and the rules can be applied. For retrieving such values, the graph transformation engine matches the pattern in the rules consisting of nodes and edges with the model graph. Because the *moduleName* parameter provides a value for a newly created class instance, it cannot be matched automatically and requires input from the user before it can be applied.

Quality. For determining the quality of the obtained modularization model, we follow an *Equal-Size Cluster Approach* (ECA) [39]. The *Equal-Size Cluster Approach* attempts to produce a modularization that contains modules of roughly equal size. Specifically, we use the following metrics for measuring the quality of the modularization: (*i*) coupling, (*ii*) cohesion, (*iii*) modularization quality (MQ), (*iv*) number of modules, and (*v*) the difference between the maximum and minimum number of classes in a module. Please note that in general several metrics may exist to measure the same aspect of a system, but the results retrieved from these metrics do not necessarily compare with each other [44]. For instance, the coupling and cohesion of a system can be calculated using several different metrics, as studied by Abdeen et al. [45]. In our example, coupling refers to the number of external dependencies a specific module has, that is, the sum of inter-relationships with other modules. This translates to the number of class dependencies (*dependsOn* in our meta-model depicted in Figure 2) that origin in one module but end in another. Cohesion refers to the dependencies within a module, that is, the sum of intra-relationships in the module. In our example, this reflects the number of class dependencies that origin and end in the same module. Typically, low coupling is preferred as this indicates that a module covers separate functionality aspects of a system, improving the maintainability, readability, and changeability of the overall system [45, 46]. On the contrary, the cohesion within one module should be maximized to ensure that a module does not contain parts that are not part of its functionality. Ideally, a module should be a provider of one functionality [45]. The calculations of coupling and cohesion used in our example are depicted in Equations 5 and 7, respectively. In these equations, M refers to the set of all modules and $C(m)$ refers to the set of classes contained in the module m.

$$\mathrm{COP} = \sum_{\substack{m_i,\, m_j \in M \\ m_i \neq m_j}} \sum_{\substack{c_i \in C(m_i) \\ c_j \in C(m_j)}} \begin{cases} 1 & \mathit{if}\, c_i \ \mathit{depends\ \ on}\ \ c_j \\ 0 & \qquad \mathit{otherwise} \end{cases} \tag{3}$$

$$\mathrm{COH} = \sum_{m_i \in M} \sum_{c_i c_j \in C(m_i)} \begin{cases} 1 & \mathit{if}\ \ c_i\ \ \mathit{depends\ \ on}\ \ c_j \\ 0 & \qquad \mathit{otherwise} \end{cases} \tag{4}$$

The MQ [39] evaluates the balance between coupling and cohesion by combining them into a single measurement. Because MQ is a well-studied objective function, we include it as an objective. This is one of the attractive aspects of multi-objective and many-objective approaches, we can always include other candidate single objectives as one of the multiple objectives to be optimized [39]. It has been proven that the higher the value of MQ, the better the quality of the modularization [47]. The aim is to reward increased cohesion with a higher MQ score and to punish increased coupling with a lower MQ score. The MQ is defined in Equation 9, where MF_m is the modularization factor for module m. The modularization factor is the ratio of intra-edges to edges in each cluster, as defined in Equation 9, where i is the number of intra-edges, that is, cohesion, and j is the number of all edges that originate or terminate in module m. The reason for the occurrence of the term $\frac{1}{2}$ rather than merely j is to split the penalty of the inter-edge across the two modules connected by that edge [39]. The MQ value will tend to increase if there are more modules in the system, so it also makes sense to include the number of modules in the modularization as an objective. Indeed, also in order to avoid having all classes in a single large module which would yield no coupling and a maximum cohesion, we maximize the number of modules. At the same time, the maximum and minimum number of classes in the modules ought to be minimized to aim at equal-sized modules.

$$\mathrm{MQ} = \sum_{m \in M} MF_m, \quad \text{where} \quad MF_m = \begin{cases} 0 & \mathit{if} \;\; i = 0 \\ \dfrac{i}{i + \frac{1}{2}j} & \mathit{if} \;\; i > 0 \end{cases} \tag{5}$$

With the rules and quality objectives presented, it is not clear how to orchestrate the two rules depicted in Figure 4 in order to obtain a desired model where all measures are optimized. In fact, model transformation languages do not offer any mechanism to specify the objectives of a transformation explicitly. Besides, if objectives were encoded somehow within the transformation, for example, by using a scheduling language for transformation rules or encoding the schedule as additional positive or negative application conditions, the transformation could only be used to solve this particular problem and only for the given objectives. On the other hand, if they are not encoded in the transformations, it is not clear how the rules affect the objectives and how they should be applied. Therefore, in next section, we show our approach to solve problems of this type.

3. TRANSFORMATION ORCHESTRATION WITH MARRYING OPTIMIZATION AND MODEL TRANSFORMATIONS

This section introduces the MOMoT approach by explaining the requirements, the approach at a glance, and subsequently, also in detail by discussing its implementation.

3.1. Requirements

As we have outlined in the previous section, model engineers are facing problems to deal with huge or even infinite state spaces which may have a multi-modal nature. The goal of our approach is to support model engineers in solving such problems which are either too large or too expensive to solve through an exhaustive or manual approach. Specifically, we provide an approach which enables model engineers to specify their problems declaratively, that is, by specifying what is expected from the output model (objectives and constraints) but not how to get there (transformation orchestration). In order to provide such an approach, the following requirements are desirable:

Generic	The approach should be problem-agnostic and algorithm-agnostic so that no assumptions about the problem or the selected search algorithm need to be made in advance. The switch between algorithms should be simple and the changes necessary to switch from one problem to another problem should be minimal to reduce the learning effort.
Transparent	In order to understand the approach, the executed process must be transparent to the user. This means that the approach must (*i*) take the artifacts produced by the model

engineer as input, that is, the model which constitutes the problem instance and the set of model transformation units that can manipulate this model, (*ii*) produce output on the same level as the input, that is, an output model conforming to a meta-model and a sequence of orchestrated transformation units, and (*iii*) provide analysis about the search process.

Declarative	The goal of the orchestration, that is, the objectives and constraints of the output model, must be separated from the model transformation specification and elevated to first-class citizens. By making the objectives explicit, in contrast to implicitly defining them as part of the model transformation semantics, we make both the transformations and the objective specifications re-usable for other problems. Additionally, explicit objectives may also serve as documentation for the problem solving process. The language(s) in which the objectives and constraints can be defined should be known by the model engineer.
Supportive	Because it cannot be assumed that every model engineer is proficient in search-based techniques, a support system must be in place to inform the model engineer about different aspects of the meta-heuristic search and prevent common mistakes when configuring the search for a specific problem.

3.2. Marrying optimization and model transformations at a glance

In order to realize an approach that meets the earlier stated requirements, we combine MDE techniques with SBSE techniques. Instead of manually deriving an orchestration of transformations for a given scenario in a specific problem domain, dedicated search algorithms are employed to calculate the transformation orchestration based on a given set of objectives and constraints. In fact, SBSE techniques allow us to address multi-modality problems as they aim to find the Pareto-optimal set of solutions, as opposed to trying to obtain a single optimal solution. For the modularization problem this would mean that we are interested in a set of solutions where all objectives are compensated and optimized instead of being combined into a single metric, which may not achieve optimality [39]. An overview of our approach is depicted in Figure 5.

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, (*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 provides feedback about the specified search configuration. Because 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 the search to find transformation orchestrations that optimize the given objectives and fulfill the specified constraints. In the following sections, we describe the different parts of our approach on the basis of the modularization problem introduced in the previous section.

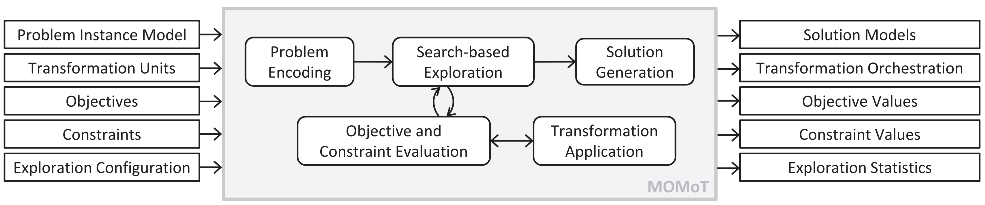


Figure 5. Overview of the MOMoT approach.

3.3. Problem encoding: meta-models and models

As it is typical in MDE, the problem domain itself is defined as a meta-model (e.g., cf. the meta-model of the modularization problem depicted in Figure 2). Based on the specific problem domain, a user can define both concrete problem instances, that is, models (e.g., cf. Figure 3) and transformation rules that specify how problem instances can be modified (e.g., cf. Figure 4) in order to produce a solution, that is, output model (e.g., cf. Figure 11).

3.4. Solution Representation: Model Transformations

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 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. 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, that is, 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. A description of all supported transformation units based on [16] can be found in Table I.

The most basic *transformation unit* is a transformation rule which directly manipulates the model being transformed. Other transformation units can be used to combine one or more transformation units to provide simple control-flow semantics. It is important to note that a loop unit may create an infinite search space in which our approach would not stop until it runs out of memory, for example, when the provided sub-unit manipulates the graph in a way that produces additional matches for that

Table I. Transformation units based on [16] that serve as decision variables in the encoding.

Unit	Description
Transformation Rule	Rules are the main transformation units and the only ones that are ultimately executed on the input model. A rule consists of a left-hand side and a right-hand side graphs which describe the pattern to be matched and the changes to be made, respectively. Rules may have positive and negative application conditions and can be nested to execute inner rules as long as possible if the outer rule matches.
Sequential Unit	A sequential unit executes all sub-units in the given order once and may be configured to stop the execution if any of the sub-units cannot be executed and to rollback any changes that have been made by the sequential unit.
Priority Unit	A priority unit executes the first given sub-unit that can be successfully applied on the given input model. Subsequent sub-units are not executed.
Independent Unit	An independent unit executes the first randomly selected sub-unit that can be executed on the given input model. Other sub-units are not executed.
Loop Unit	A loop unit consists of one sub-unit. The loop unit is executed as long as the sub-unit can be applied in the underlying graph.
Iterated Unit	An iterated unit is composed of one sub-unit that is executed as often as explicitly specified.
Conditional Unit	This unit enables the expression of an if-then-else condition. It consists of at least two sub-units (*if*, *then*) and an optional third sub-unit (*else*). The *then*-unit is only executed if the *if*-unit can be successfully matched, otherwise the *else*-unit is executed if possible.
Placeholder Unit	This unit acts as a placeholder within the transformation without modifying the given input model.

sub-unit. Currently, we have no way to automatically detect whether a loop unit creates an infinite search space, but we inform the user about the possibility using our support system (cf. Section 3.8).

Each transformation unit can have an arbitrary number of parameters that need to be instantiated, that is, values for these parameters must be found. Additionally, application conditions can be specified that need to be fulfilled in order for the unit to be applicable. As typical for graph transformation systems, there are two kinds of application conditions, positive application conditions (PACs) and negative application conditions (NACs). The former require the presence of certain elements or relationships in the model, whereas the latter forbid their presence. Besides using model elements and relationships in the PACs and NACs, it is also possible to formulate these conditions using parameter values or attributes of elements. In such a case JavaScript or OCL may be used. As a special case, we also allow the use of transformation unit *placeholders*, that is, units that are not actually executed and do not have any effect on the output model. This enables the actual solution length to vary in cases where the solution length must be fixed in advance. Besides the decision variables, the objective and constraint values, we also provide attributes of a solution which serve as a key-value storage for additional information relating to the evaluation or to the search process.

In summary, one solution consists of an ordered sequence of applicable transformation units (decision variables) that creates a valid output model when executed on the specified input model. An example of a solution for the modularization problem with the first two transformation units and one placeholder is depicted in Figure 6.

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 rules provided for the modularization problem (cf. Figure 4), the *class* and *module* of the *assignClass* rule are matched parameters, whereas the *moduleName* parameter of the *createModule* rule is a user parameter.

Values for the matched parameters can be retrieved from the respective transformation engine through a matching procedure. In our approach, we use non-deterministic matching, that is, for a given model a different transformation unit with different parameters may be selected each time the matching is executed. This leads to a non-deterministic optimization, where we produce different results every time the search is executed. However, during the search, the matching parameters within a solution are fixed, unless otherwise modified, that is, a solution always produces the same output model for the given input model. Values for the user parameters need to be handled differently, as a user usually can provide a value for these programmatically or via a dedicated user interface when the unit is applied manually. In an automated approach, however, we need a way to generate those values when needed. This is also necessary in order to facilitate the creation of random solutions and populations, what is needed by most search algorithms. Usually, a high variance of parameter values is preferred to cover as much area of the search space as possible. By default, we use random parameter value generators for most primitive values. The range of these value generators is the range of the data type of the respective parameter, for example, 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 configuration to prune such unfruitful areas of the search space in advance.

Decision Variables (Transformation Units)				Objectives	Constraints
unit = createModule	unit = assignClass			Coupling = 66.0	UnassignedClasses = 0
moduleName = 'ModuleZrldp'	module = ModuleZrldp			Cohesion = -24.0	EmptyModules = 0
	class = FreeInode			MQ = -1.964	
		Placeholder	...	MinMaxDiff = 4.0	
				NrModules = -6.0	
				Length = 33.0	

Figure 6. One solution with the first two transformation units and one placeholder.

Additionally, we provide a hook where a user can integrate how values should be generated for specific parameters. Furthermore, the user may define which parameters should be retained as part of the solution. By default all parameters are kept, any other parameters are re-matched by the graph transformation engine when the respective unit is executed again.

Solution Repair. Even though constraints can be used to specify the validity or feasibility of solutions, a solution that is the product of re-combining two other solutions (such as needed in evolutionary algorithms) might have unit applications that are no longer executable. By default, units that cannot be executed are ignored. However, this behavior might not be satisfactory in some cases as the process of establishing that a transformation unit cannot be executed is quite expensive due to unnecessary match finding performed 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. Which strategy is the best depends on the problem at hand. Replacing a non-executable unit application with a placeholder effectively shortens the solution length and has the risk of removing useful applications from the search space which may cause the search to converge too early. On the other hand, replacing a non-executable unit application with a random executable unit application may have an impact on the overall solution quality as other transformation units might become non-executable due to the new transformation unit and may also need to be replaced. The resulting solution may be a solution that is quite different from the original solution, which may negate the positive effect of selection and recombination. Therefore, this strategy should be used with caution, especially when many different transformation rules are given and the risk of producing a very different solution is higher.

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 re-combination operator that is able to consider some constraints, for example, the partially matched crossover (PMX) [48] can preserve the order of variables within a solution.

3.5. Solution fitness: objectives and constraints

As described in Section 2.1, 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, for example, minimizing the coupling of modularization models, and objectives that relate to the solution encoding, for example, minimizing the number of transformations that should be applied to reach a solution. Additionally, a solution candidate may be subject 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, for example, all classes must be assigned to a module, and solution-specific constraints, for example, a specific transformation rule needs to be applied at least once.

Objectives and constraints can be defined either by providing a direct implementation of the respective objective value or by specifying model queries in OCL. In our approach, the support for OCL is crucial as model engineers are typically proficient in OCL and therefore can provide the necessary input in a language they are familiar with. Alternatively, users may also provide objectives and constraints in a Java-like expression language in case they are not familiar with OCL. Constraints may also be defined as NACs directly in the transformation units as we are building upon the expressive power of a graph transformation system. 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.

In summary, the problem domain is represented as a meta-model, from which a concrete problem instance model can be created. Transformation units defined upon concepts of the meta-model are then used to modify the model instances. The objectives that should be optimized in order to create the desired output model may be defined directly or via model queries in OCL. The constraints that a possible solution must fulfill in order to be valid can also be specified in OCL, but may also be encoded as NACs directly in the rules.

3.6. Exploration configuration

In order to search for good solutions to the given search problem, the user must select and configure at least one algorithm and specify the parameters of the experiment configuration.

Algorithm Selection. As mentioned previously, our proposed approach is independent from specific search algorithms. Nevertheless, additional algorithm-specific exploration options need to be configured by the user. These options depend on whether an evolutionary algorithm or a local search algorithm is used.

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 *k* 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, that is, 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 transformation application units for which many generic operators already exist, for example, the one-point crossover operator that 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 units into account, we have introduced three dedicated mutation operators. 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 (cf. Section 3.4). The third operator selects a random position within a solution and replaces all transformation units after that position with a random, executable transformation unit.

Local search algorithms maintain one solution at a time and try to improve that solution in each iteration. Improvement depends on the *solution comparison method* the user selects, for example, 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, that is, a solution that is a slight variation of the current solution. The calculation of neighbors from the current solution can be performed 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 approach, we support two neighborhood functions. Following the principle of reuse, 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, for example, when varying floating point rule parameter values. Therefore, an upper bound on the number of calculated neighbors can be specified. The second neighborhood function adds an additional, random transformation unit to the current solution, increasing its solution length. Here, an upper bound on the solution length can be specified.

If multiple algorithms are configured by the user, each algorithm is executed individually and their results can be used separately or in combination (Section 3.7).

Experiment. In order to execute the search for the selected and configured algorithms, the user needs to provide additional experiment parameters. The first parameter is the *population size*, that is, the number of solutions in each iteration for evolutionary algorithms. Second, the user must specify a stopping criterion for the algorithms, that is, the *maximum number of fitness evaluations* that are performed by each algorithm. The number of iterations of an algorithm is implicitly calculated by dividing the maximum number of fitness evaluations by the population size. Finally, the user needs to specify the number of times each algorithm should be executed (*number of runs*). As explained by Harman et al. [49], experiments are typically repeated 30-50 times in order to draw statistically valid conclusions from the results.

As part of the optional configuration parameters, a user may specify a reference set of Pareto-optimal solutions, that is, a set of known, good solutions, to enable the analysis of the search process (cf. Section 3.7). Additionally, a user may configure our approach to print detailed information about the on-going search process on the console or terminate a single run of an algorithm based on different criteria, for example, after a certain time limit has been reached or a given solution has been found.

As our target audience is model engineers, we aim to make the configuration of the exploration easy and provide support with respect to their configuration options (cf. Section 3.8).

3.7. Result analysis

Solutions. Finally, after the experiment with the configured search algorithms has been executed, the user can process the results. As results, we have (*i*) the set of orchestrated transformation sequences leading to (*ii*) the set of Pareto-optimal output models with (*iii*) their respective objective values. Because the results are on the level of the provided input, that is, transformation rules, the set of orchestrated transformation sequences can be inspected by the model engineer to get an understanding of the produced results. The set of output models produced by these transformations conform to the same meta-model as the given input model and can be inspected or processed further by MDE tools. The objective values may give an overview of how well the objectives are optimized by the different algorithms. All results of the experiment can be retrieved for each algorithm individually or combined into one set of Pareto-optimal solutions. Especially, the objectives of the combined solutions may be of interest, as they can be used as a reference set for future experiments if the Pareto set of optimal solutions is not known a priori, which is the case for many real-world problems. Individual algorithms results may be of interest if different algorithms or different configurations of one algorithm need to be compared.

Statistical Analysis. Collected runtime data of the search process can be used to plot graphs in order to give a better overview about the algorithm executions and performance. For example, Figure 7 depicts the convergence of the MQ objective over time for multiple runs of two local search algorithms, Hill Climbing and Random Descent. Each line corresponds to one run of the respective algorithm.

If the user provides a reference set of solutions, we can also calculate several *indicators* during the search. These indicators are used to gain insight into the search performance of the different

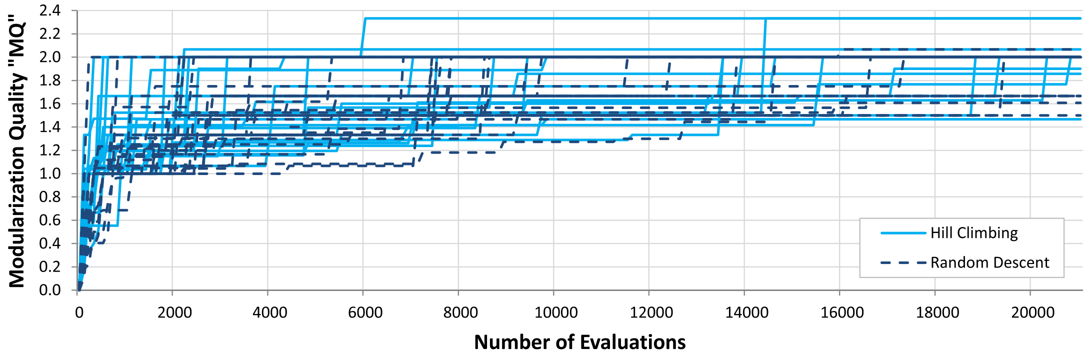


Figure 7. Example graph of modularization quality convergence for local search.

algorithms. As an example, we describe two common indicators, Hypervolume and inverted generational distance. 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. Hypervolume can capture both the convergence and the diversity of the solutions. Inverted generational distance is a convergence measure that corresponds to the average Euclidean distance between the Pareto set produced by the algorithm and the given reference set. We can calculate the distance between these two fronts in an M-objective space as the average M-dimensional Euclidean distance between each solution in the approximation and its nearest neighbor in the reference front. Better convergence is indicated by lower values. For each run of an algorithm, we can calculate one value per indicator.

In order to draw valid conclusions from these values, we need to perform a statistical analysis, for example, using the Mann–Whitney U test [50]. The Mann–Whitney U test, equivalent to the Wilcoxon rank-sum test, is a non-parametric test that allows two solution sets to be compared without making the assumption that values are normally distributed. Specifically, we test under a given significance level α the null hypothesis (H_0) that two sets have the same median against the alternative hypothesis (H_1) that they have different medians. If the resulting p-value is less than or equal to α, we accept H_1 and we reject H_0; if the p-value is strictly greater than α, we do the opposite. Usually, a significance level of 95% ($\alpha = 0.05$) or 99% ($\alpha = 0.01$) are used. Alternatively, in our approach, the user may choose another test like Kruskal-Wallis One-Way Analysis of Variance by Ranks test [51, 52].

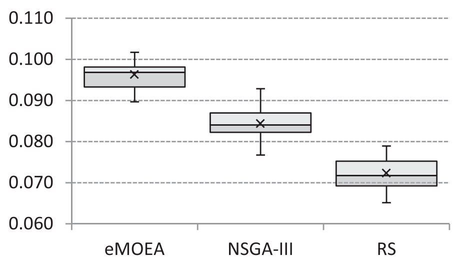
Using these statistical tests, we can determine whether two algorithms are interchangeable with respect to a certain indicator. However, it is not possible to show the magnitude of how much one algorithm is better than another, that is, the *effect size*. In our approach, we support parametric effect size measures with Cohen's d statistic [53] and non-parametric effect size estimates with Vargha and Delaney's A measure [54] and Cliff's delta [55]. For the non-parametric Mann–Whitney U test, we may use a non-parametric effect size measure such as Cliff's delta. Cliff's delta makes no assumption about the distribution or spread of the compared value vectors. The calculation of Cliff's delta d is based on the average number of times (#) a value x_i of a vector of size n is larger or smaller than a value x_j of a vector of size m, as depicted in Equation 11. The resulting effect size ranges between -1 and $+1$ or between 0 and 1, if we consider the absolute value, whereas a value of 0 represents a complete overlap of both vectors and a value towards 1 represents an increasing difference between the two vectors. The effect size can be considered (*i*) small if $0.147 \leq d < 0.33$, (*ii*) medium if $0.33 \leq d < 0.474$, and (*iii*) large if $d \geq 0.474$ [56].

$$d = \frac{\#(x_i > x_j) - \#(x_i < x_j)}{mn} \tag{6}$$

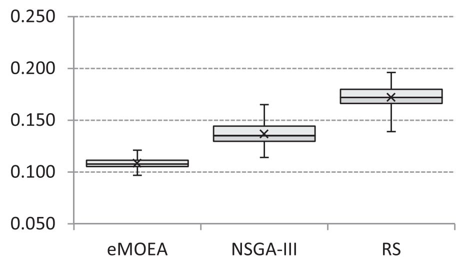
As an example experiment, we use three algorithms, ε-MOEA algorithm, NSGA-III, and Random Search (RS), and execute each algorithm 30 times on the mtunis system with a population size of 100 and for 200 iterations. We use tournament selection with $k = 2$, a one-point crossover operator and a placeholder mutation operator with $p = 0.1$. The results of the analysis are shown in Table II, in which the best values for each indicator are highlighted in bold, and depicted in Figure 8. Each box plot in the figure shows the minimum value of the indicator (shown by the lower whisker), the maximum value of the indicator (shown by the upper whisker), the second quantile (lower box), the third quantile (upper box), the median value (horizontal line separating the boxes), and the mean value of the indicator (marked by an x) for each algorithm. We can clearly see that for the Hypervolume indicator, RS has 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. In order to investigate the results further, we deploy the Mann–Whitney U test with a significance level of 99% as described earlier. Based on this statistical analysis, we determine that the results of all algorithms are statistically different from each other, that is, no two algorithms perform equally well. This is also indicated in Table II (row *indifferent*) as empty set. Finally, we calculate the effect size and find the magnitude of the differences between the algorithms using Cliffs's delta measure. From the results, we can see that all differences are considered large. The

Table II. Excerpt of indicator statistic for different multi-objective algorithms.

	Hypervolume			*Inverted Generational Distance*		
	ε -MOEA	NSGA-III	RandomSearch	ε -MOEA	NSGA-III	RandomSearch
Min	**0.089**	0.078	0.065	**0.097**	0.114	0.139
Median	**0.097**	0.084	0.072	**0.108**	0.135	0.172
Max	**0.102**	0.093	0.079	**0.121**	0.165	0.196
Mean	**0.097**	0.084	0.072	**0.108**	0.136	0.172
StdDev	0.003	0.004	0.004	0.006	0.011	0.011
Count	30	30	30	30	30	30
Indifferent	{}	{}	{}	{}	{}	{}



(a) Hypervolume



(b) Inverted Generational Distance

Figure 8. Hypervolume and Inverted Generational Distance box plots for the modularization problem.

effect size for Hypervolume between ε -MOEA and NSGA-III is $d = 0.9867$, between NSGA-III and RandomSearch it is $d = 0.98$, and between ε -MOEA and RS it is $d = 1.0$. A similar effect size is calculated for the inverted generational distance. As a conclusion, we can state that for the settings we have chosen and based on our selected performance indicators, ε -MOEA performs the best, NSGA-III performs second best, and RS has the worst performance. The fact that a sophisticated meta-heuristic search outperforms RS is also a good indicator that the problem is suitable for SBSE techniques.

3.8. Support system

Configuration Language. To support the model engineer in specifying the search problem, the necessary input and the experiment configurations, we provide a dedicated configuration language with MOMoT. Conceptually, this domain-specific language (DSL) is defined platform-independently using a meta-model and an expression language. Thus, the specified configurations are represented as models in MOMoT. In fact, all configurations can be made in a single file with a few lines of mandatory configuration parameters and some optional configuration parameters. In general, the textual notation of the language uses '=' to assign values, '{}' to define objects, '[]' to denote lists and ':' to separate key-value pairs. Content-Assist provides the next allowed configuration parameters and guides the users when they need to provide their own input. Furthermore, when the user is required to implement specific behavior, we provide helper objects with additional convenience methods. An excerpt of a configuration is depicted in Listing 1. Using such configuration, we can either generate code or interpret the configuration directly to execute the search.

Static Checks. One advantage of providing a DSL over providing just an API is that we can statically check the given configurations before we compile and execute them. These static checks can be used to *inform* the user about certain aspects of the search, *warn* about uncommon configurations, and prohibit the execution in case any *errors* can be detected. These checks are integrated in the editor of the DSL providing the details on the respective positions in the text. Additionally, an overview is used to show all information in a list. A summary of all provided checks is shown in Table III. The upper part of the table shows checks related to the search, while the lower part shows consistency checks. Dependencies

Table III. List of static checks in the configuration DSL.

Name	Type	Description
Unit Applicability	Error	Check if any of the given transformation units can be applied.
User Parameters	Error	Check if values are provided for user parameters.
Algorithm Runs	Warning	Warn if the number of runs is smaller than the recommended minimum (< 30) [49].
Number of Iterations	Warning	Warn if the resulting number of iterations is very small (≤ 10).
Many-Objective	Warning	Warn if we have many-objective search (> 3 objectives) [25] but a multi-objective or local search algorithm has been chosen.
Object Identity	Warning	Warn about missing `equals` implementation for parameters.
Population Size	Warning	Warn if the population size is very small (≤ 10).
Algorithm Parameters	Info	Inform about the relevance of different algorithm parameters, e.g., about specific evolutionary operators.
Loop Unit	Info	Inform the user about the possibility of a potential infinite search space created through a loop unit.
Single-Objective	Info	Inform about local search algorithms if only a single objective is used.
Algorithm Name	Error	Check if every algorithm name is unique.
Input Model	Error	Check if the specified input model exists.
OCL Dimension	Error	Check if the given dimension are valid OCL queries.
Parameter Value	Error	Check if parameter values are defined for existing parameters.
Parameter Value Name	Error	Check if parameter values are not defined twice for the same parameter.
Reference Set	Error	Check if the given reference set file exists.
Transformations	Error	Check if the specified transformations exist.
Save Analysis	Warning	Warn if analysis should be saved but has not been defined.
Result Analysis	Info	Inform when an existing analysis file will be overridden.
Result Objectives	Info	Inform when an existing objectives file will be overridden.

between the individual configurations are considered by the editor and invalid configurations are marked and prevent execution. Please note that some of these checks are more complex than others, for example, checking if any of the given transformation units can be applied can only be answered if we actually load the model and transformations and try it out. Therefore, not all checks are performed on the fly and the user needs to explicitly trigger the more complex validation process.

Listing 1: Excerpt of an experiment configuration in the textual notation of our DSL

```
model = "modularization_model.xmi" // has a reference to the meta-model
transformations = {modules = ["modularization.henshin"]}
fitness = {
 objectives = {…
   NrModules : maximize "self.modules->size()" // OCL-specification
   Length : minimize new TransformationLengthDimension // generic objective
 }
 constraints = {…
   UnassignedFeatures : minimize {// Java-like syntax, direct calculation
      (root as ModularizationModel).classes.filter[clc.module == null].size}
 }
}
algorithms = {… // moea is an auto-injected object
 Random : moea.createRandomSearch()
 NSGAIII : moea.createNSGAIII(
   new TournamentSelection(2), // k == 2
   new OnePointCrossover(1.0), // 1.0 == 100%, always do crossover
   new TransformationPlaceholderMutation(0.15)) // 15% mutation
}
experiment = {
 populationSize = 300
 maxEvaluations = 21000
 nrRuns = 30
 referenceSet = "reference_objectives.pf" // if available (optional)
 collectors = [hypervolume] // collect during search, needs referenceSet
}
results = {
 objectives = {outputFile = "data/output/all/modularization_model.pf"}
 models {
   algorithms = [NSGAIII]
   outputDirectory = "data/output/nsgaiii/modularization_model/"}
}
```

3.9. Implementation

In order to show the feasibility of our approach, we provide an implementation of MOMoT as a Java-based framework. While it is possible to implement the approach from scratch, reusing the functionality of existing frameworks as much as possible is the central principle of our implementation. Reuse 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 existing frameworks. Specifically, to unify the MDE and SBSE worlds in a single framework, we bridge the EMF, the Henshin graph transformation system, and the MOEA Framework. The complete code with further explanations as well as the examples presented in this paper can be found on our project website [57].

Eclipse Modeling Framework. Eclipse Modeling Framework is an open-source, Eclipse-based framework that supports the creation of modeling and meta-modeling tools. 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. Particularly, we use Eclipse OCL to evaluate OCL queries and constraints on Ecore-based models, Xbase[5] as a base language for our configuration language which already provides code generation and interpretation capabilities, and the EMF Validation Framework to implement the configuration checks for our configuration DSL.

Henshin. As graph transformation language and engine in our implementation, we use Henshin [16], because it offers a rich language and associated tool set for in-place transformations of Ecore-based models. In Henshin, graphs are attributed, and nodes, edges, and attributes refer to *EClass*, *EReference*, and *EAttribute* of Ecore, respectively. 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. An example of the visual notation of that language has been shown in Figure 4. Besides rules, Henshin provides control-flow units to orchestrate these rules, for example, sequential units, priority units, or amalgamation units [58]. In order to tackle the graph pattern matching problem, Henshin uses a technique of constraint solving [59] where each node in the graph is considered a variable with a corresponding domain slot that is the solution space for that variable. Impossible solutions for each variable are then removed according to the given constraints. The constraints are extracted from the provided patterns and the meta-model and are categorized into different groups, for example, type constraints, reference constraints, or containment constraints. To improve performance, the order in which variables and constraints are processed is determined heuristically on-the-fly based on the current graph. The final matches are constructed by locking variables to specific solutions after the impossible solutions have been removed.

MOEA Framework. The MOEA framework is an open-source Java library which provides a set of multi-objective evolutionary algorithms with additional analytical performance measures and which can be easily extended with new algorithms. By reusing the MOEA framework, we can use the following

[5]Eclipse Xbase. Available at https://wiki.eclipse.org/Xbase

evolutionary algorithms out of the box: NSGA-II, eNSGA-II, NSGA-III, eMOEA, and RS. Furthermore a set of selection and crossover operators are provided, which can also be reused. A user may choose to develop further algorithms or integrate existing ones from the jMetal library[6], the PISA library[7] and the BORG MOEA Framework[8], for which adapters or specific plug-ins are already provided by MOEA. In order to support also local search algorithms, we provide a base interface and implementation as well as the described neighborhood functions. We show the use of our local search hooks by implementing the Random Descent and Hill Climbing algorithms as proof-of-concept.

4. EVALUATION

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

RQ1 *Applicability*: Is our approach applicable to challenging problems in model-based software engineering?
RQ2 *Overhead*: How much runtime overhead is introduced by our approach compared with a native encoded solution?
RQ3 *Search Features*: What are the additional search features offered by MOMoT w.r.t. pure transformation approaches?

In order to answer RQ1 and RQ2, we use four case studies that target different problem areas and differ in their level of complexity with regard to the transformation implementation. The answer to RQ3 is given by argumentation based on the analysis of the default Henshin transformation engine compared with the MOMoT extension. Thus, this discussion is valid for all case studies.

4.1. Case study setup

This section explains the four case studies used to evaluate our approach and framework with respect to the RQs. The first case study is known as the stack balancing example, and is used to exemplify a simple problem domain and to show that our approach is not only applicable to software systems, but to any system where MDE is used in its construction process. The remaining case studies represent classical problems of model-based software engineering.

Stack Load Balancing. The problem domain is a set of stacks that are inter-connected in a circular way, where each stack can have a different number of boxes referred to as *load*. The main goal of this study is to find a good load balance. The two objectives that should be minimized are the *standard deviation of the loads* and the *solution length*.

Class Modularization. This is a classic problem in software architecture design that is used as running example in this paper (cf. Section 2.4). The goal is to group a number of classes that have inter-dependencies into modules in order to optimize five objectives described previously. Among these objectives, coupling and cohesion are two well-known conflicting objectives. A problem instance of this case study consists of a set of classes and their inter-dependencies, as shown in Figure 3 for the *mtunis* system [43]. To manipulate instances of this kind, we need to (*i*) create new modules and (*ii*) assign classes to existing modules. A valid solution assigns each class to exactly one module and has no empty modules.

Many formulations of this problem have been tackled using search-based optimization methods. For instance, Mancoridis et al. [60] presented the first search-based approach to address the problem of software modularization using a single-objective approach. The same technique has also been used by Mitchell and Mancoridis [61, 62] using coupling and cohesion as objectives. Harman et al. [63] use a genetic algorithm to improve the subsystem decomposition of a software system by considering a

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

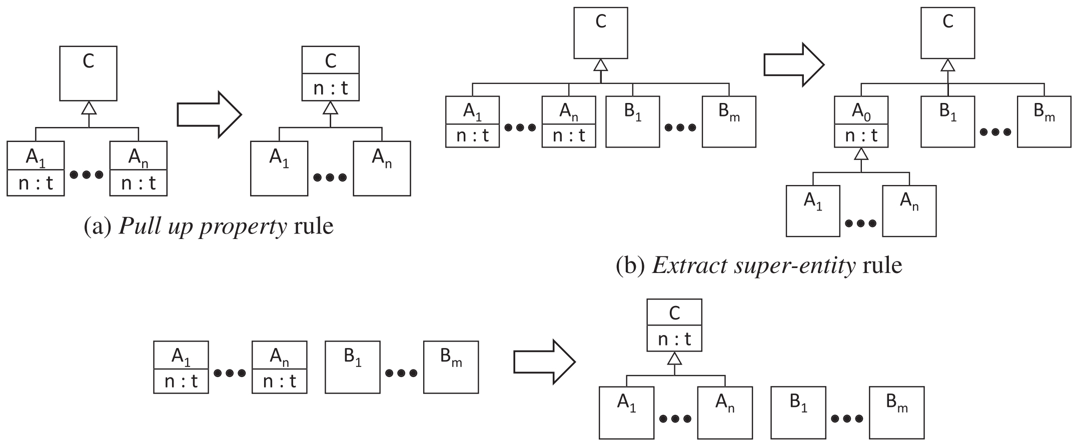
combination of quality metrics, for example, coupling, cohesion, and complexity, in the fitness function. Abdeen et al. [64] proposed a heuristic search-based approach to automatically reduce the dependencies between packages of a software system using simulated annealing based on moving classes between packages. In [39], Praditwong et al. formulated the software clustering problem as a multi-objective optimization problem. Their work aims at maximizing the MQ measurement, minimizing the inter-package dependencies, increasing intra-package dependencies, maximizing the number of clusters having similar sizes, and minimizing the number of isolated clusters. Following a similar idea, Mkaouer et al. [65] proposed to remodularize object-oriented software systems using many-objective optimization with seven objectives based on structural metrics and history of changes at the code level.

Class Diagram Restructuring. Refactoring is a technique tailored at enhancing object-oriented software designs through the application of behavior-preserving operations [66]. However, there can be a high number of choices and complex dependencies and conflicts between them that makes it difficult to choose an optimal sequence of refactoring steps that would maximize the quality of the resulting design and minimize the cost of the transformation [67]. Refactoring can become very complex when we deal with large systems because existing tools offer only limited support for their automated application [68]. Therefore, search-based approaches have been suggested in order to provide automation in discovering appropriate refactoring sequences [40, 69–71]. Thereby, the idea is to see the design process as a combinatorial optimization problem attempting to derive the best solution, with respect to a given objective function, from a given initial design [67, 72].

Our third case study is taken from the Transformation Tool Contest (TTC) of 2013 [73]. The aim of the Transformation Tool Contest series is to compare the expressiveness, the usability, and the performance of graph transformation tools along a number of selected software engineering case studies. Specifically, we use the class diagram restructuring case study [74, 75], which consists of an in-place refactoring transformation on UML class diagrams realized in the design or maintenance phase of a system. A problem instance consists of a set of entities, for example, classes, and their properties, for example, attributes or methods, as well as possible inheritance relationships between the entities (we only consider single inheritance in this case study). The goal is to remove duplicate properties from the overall class diagram, and to identify new entities which abstract data features shared in a group of entities in order to minimize the number of elements in the class diagram, that is, entities and properties. The three ways used to achieve this objective are (*a*) *pulling up* common properties of all direct sub-entities into a super-entity, (*b*) *extracting a super-entity* for duplicated properties of entities that already have a super-entity, and (*c*) *creating a root entity* for duplicated properties of entities that have no super-entity. They are graphically shown in Figure 9.

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 fact, considering only the number of elements in the class diagram as objective and the three rules, we can give a canonical solution that always yields the best result. First, properties need to be pulled up (rule a), then super entities should be extracted (rule b), and then new root entities should be created (rule c). If the same rule can be applied on more than one property, it must be applied in the one that has more occurrences first. The way the approaches described in [75] deal with the rule execution ordering problem is by implementing a rule prioritization mechanism when writing the transformations. In our approach, this can be performed using the *Priority Unit* (cf. Table I). Therefore, we can express the class diagram restructuring case study without any problem.

However, if such a clear ordering of rule applications can be given in advance, a search-based approach is not suitable and the problem should be solved as indicated. In order to make the case study interesting for meta-heuristic search, we extend it with an additional domain-specific objective: the *minimization of the maximal depth of the inheritance tree* (DIT) [76], that is, we aim to minimize the longest path from an entity to the root of the hierarchy. The DIT is a metric that can be used to measure the structural complexity of class diagrams [77] and the deeper the hierarchy, the greater the number of properties an entity is likely to inherit, and thus, making it more complex to predict its behavior yielding a higher design complexity. However, the deeper a particular entity is in the hierarchy, the greater the potential reuse of inherited entities [76]. Therefore, we have a conflict between the DIT and the objective to minimize the overall number of elements in the class diagram.



(a) *Pull up property* rule

(b) *Extract super-entity* rule

(c) *Create root entity* rule

Figure 9. Transformations of the class diagram restructuring case study (from [75]).

Eclipse Modeling Framework Refactor. Eclipse Modeling Framework Refactor is an open-source Eclipse project [78, 79] that supports a structured model quality assurance process for Ecore and UML models. In particular, EMF Refactor builds upon the following three concepts:MetricsSeveral metrics can be calculated on model-level to gain insight into the model, for example, average number of parameters of an operation of a class or the depth of the inheritance tree.
SmellsSmells are anti-patterns that may be an indicator for insufficient quality in the models. These smells can be generally defined, for example, the model contains an abstract class without any concrete subclasses, or based on the detected model metrics, for example, the model contains an operation with more input parameters than the specified limit (*Long Parameter List*).
RefactoringsRefactorings are changes made on model-level with the aim of improving the quality of that model. For example, to remove the smell of the *Long Parameter List* we can group a set of parameters together to introduce a separate parameter object which is used instead of the list of parameters.

4.2. *Measures*

To assess the applicability of our approach (*RQ1*), we use all four 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, that is, using meta-models and graph transformation rules, are expressive enough for real-world problems. Then, 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 modularization and the stack case studies as representatives of differently sized problems by varying the population size parameter. Finally, in order to demonstrate how our approach advances the current state of research w.r.t. search features offered by existing MDE frameworks (RQ3), we compare the search features of the pure Henshin transformation engine with the search features contributed by MOMoT.

4.3. *Results*

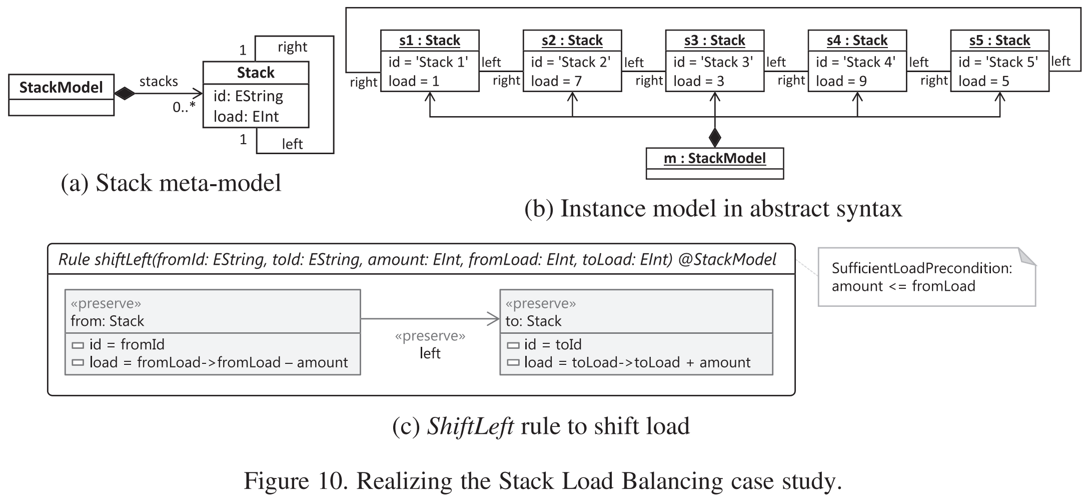
This section describes the experiments conducted with the four case studies. Based on the obtained results, we discuss the answers to our research questions. All results are based on 30 runs of the NSGA-III algorithm, which we deem sufficient for our overhead analysis because it is also the recommended minimum to deal with the stochastic nature of meta-heuristic search [49]. To ensure the sanity of the solutions found during these runs, we manually inspected the solutions for any constraint violations or contradicting objective values. Additionally, we selected one solution for each case study selected using a knee point strategy [80] to evaluate the quality of the solutions. The artifacts created for this evaluation are described in the following sections and are available on our project website [57].

RQ1. In order to answer the first research question about whether our approach is applicable to challenging problems in model-based software engineering, we modeled the problem domains of all case studies as Ecore meta-models and developed transformations rules that manipulate instances of these meta-models in Henshin.

The Stack Load Balancing case study has been modeled as shown in Figure 10. The meta-model that represents the system is depicted in Figure 10a. 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 10b. To manipulate instance models such as the one shown, we propose two basic rules to shift part of the load from one stack either to the left or to the right neighbor. The *ShiftLeft* rule is shown in Figure 10c, and an analogous rule is used to shift parts to the right. The rule contains a precondition, which ensures that the amount that is shifted is not larger than the load of the source stack, avoiding a negative load in the source stack. This attribute condition is shown as an annotation in the figure, although it has been expressed using JavaScript. Applying our approach on the input model shown in Figure 10b, we quickly find the shortest sequence of three rule applications that leaves five pieces in each stack.

The Class Modularization case study has been shown throughout the two previous sections. Thus, we keep the discussion in this section short. Section 2 describes how the problem is modeled and represented in Henshin, while Section 3 explains how it can be solved with MOMoT, and an excerpt of the necessary configuration is shown. We have applied our approach to the *mtunis* system instance consisting of 20 classes, as depicted in Figure 3. As result, we have retrieved many Pareto-optimal solutions. One of these solutions is illustrated in Figure 11, showing the respective fitness values for coupling, cohesion, the modularization quality, the difference between the minimum and maximum number of classes in a module, and the number of modules. To foster the readability of the figure, arrows inside the same module are green while arrows targeting a different module are colored according to their source module.

The Class Diagram Restructuring case study is the most complex one with regard to rule complexity. Figure 12 depicts the meta-model to which problem instances have to conform [75]. In this meta-model, the inheritance relationship among entities is modeled by using the *Generalization* concept. This way, if an entity has a super type, then the entity has a *generalization* that, in turns, points to a *general* entity. In the same way, if an entity has a child, then it has a *specialization* that, in turns, points to a *specific* entity. Please note that we deal with single inheritance, what is modeled by the cardinality *0..1* of relationship *generalization*. In fact, the problem with multiple inheritances is not considered a search problem and has been solved. For instance, Godin and Mili [81] use formal concept analysis and pruned lattices to solve the problem. These techniques are able to reach a unique canonical solution in a polynomial time [82].



(a) Stack meta-model

(b) Instance model in abstract syntax

(c) *ShiftLeft* rule to shift load

Figure 10. Realizing the Stack Load Balancing case study.

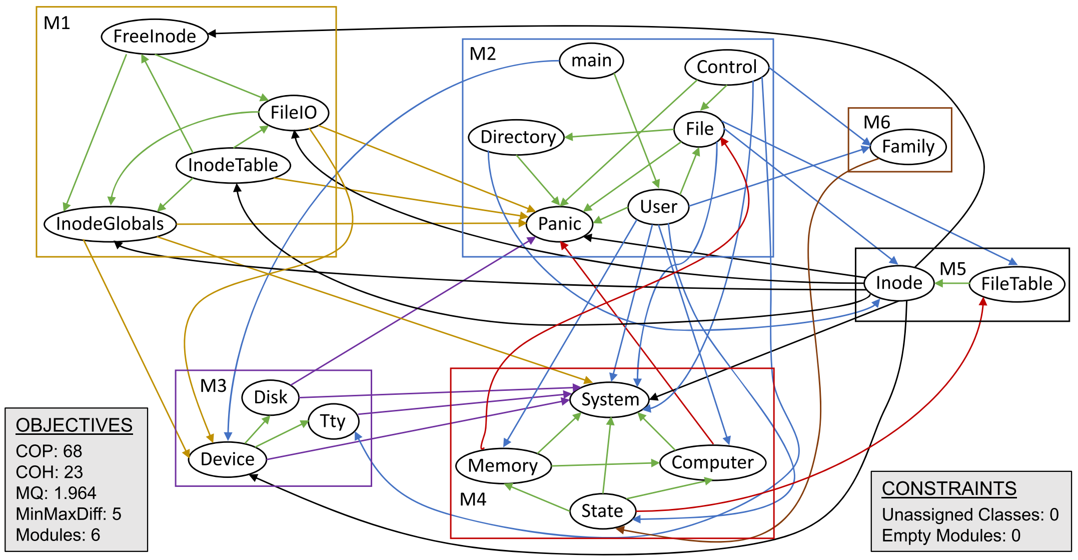


Figure 11. Modularization model: sample output instance for the *mtunis* system.

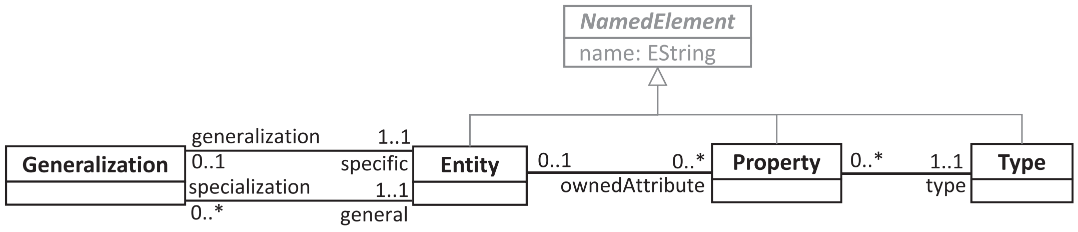


Figure 12. Metamodel of the Class Diagram Restructuring case study

In order to tackle the class diagram restructuring case study with single inheritance, we translate all three operations (cf. Figure 9) into graph transformation rules. Each rule needs at least one NAC and contains at least one nested rule. To give an estimate of the complexity to develop such a rule, we depict the rule for creating root classes in Figure 13. A nested rule (indicated by a '*' in the action name in Figure 13) 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 13) and as OCL constraints. For instance, the NAC of pulling up attributes shown in Listing 2 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 2: OCL-NAC for pulling up attributes specified in the context of the class diagram

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

[language = OCL20, caption = OCL-NAC for pulling up attributes specified in the context of the class diagram,label = lst:oclNAC] self.entities- > select(e | e.name = eName).specialization - > collect(g | g.specific) - > forAll(e | e.ownedAttribute- > exists(p | p.name = pName)) The two objectives for the fitness functions can be translated to OCL or simply calculated in Java. The calculation of the first objective, that is, the number of elements in the class diagram, is depicted in Listing 3. For the calculation of the maximum hierarchy depth, we simply traverse the paths from

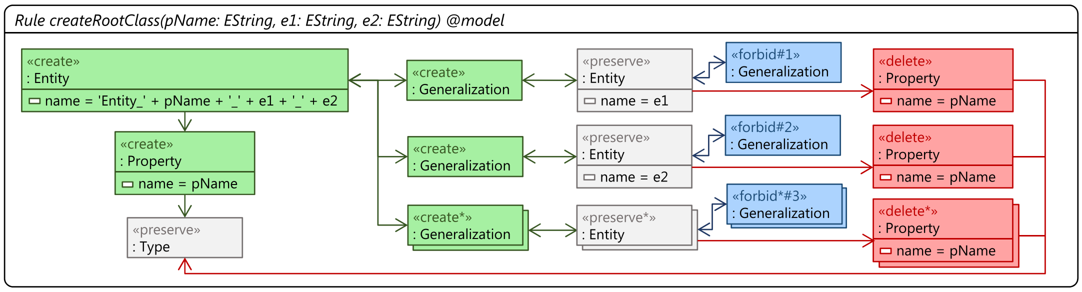


Figure 13. Create a root class for a common attribute with three NACs (*forbid*) and one nested rule ('∗').

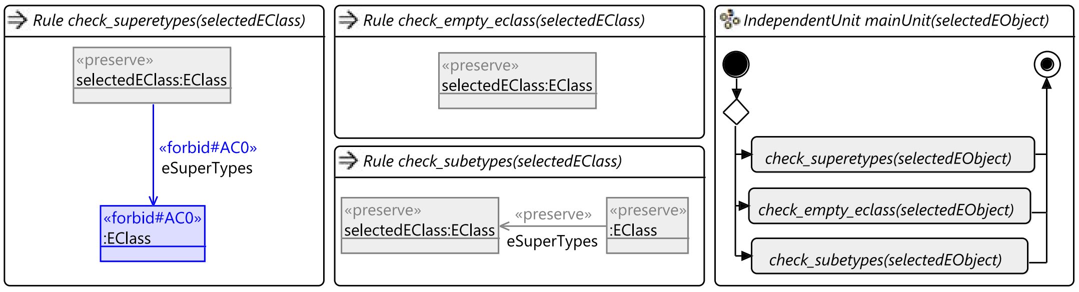
all leaf entities to their root entity with a derived property called *allSuperEntities* and determine the maximum length as also shown in Listing 3.

Listing 3: OCL expressions needed for the Class Diagram Refactoring case study

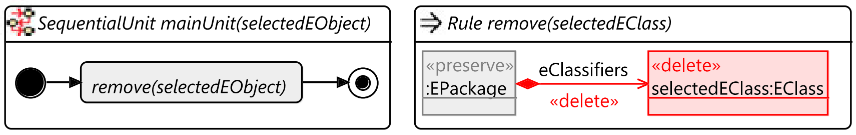
```
– computing the number of elements
self.properties–>size() + self.entities–>size()
– computing the maximum hierarchy depth
self.entities–> collect(ele.allSuperEntities –> size()) –> max()
```

Concluding, we can state, that we represent the class diagram restructuring as described in [74, 75]. In addition, we can easily handle the extension with the additional objective which would require a redesign of the other solutions for this case study as presented in [75].

In the EMF Refactor case study, the quality assurance process is partly realized with Henshin to detect metrics by counting how often a given rule can be matched, detecting smells through


Rule check_superetypes(selectedEClass)
«preserve»
selectedEClass:EClass
«forbid#AC0»
eSuperTypes
«forbid#AC0»
:EClass
Rule check_empty_eclass(selectedEClass)
«preserve»
selectedEClass:EClass
Rule check_subetypes(selectedEClass)
«preserve»
selectedEClass:EClass
«preserve»
eSuperTypes
«preserve»
:EClass
IndependentUnit mainUnit(selectedEObject)
check_superetypes(selectedEObject)
check_empty_eclass(selectedEObject)
check_subetypes(selectedEObject)

(a) Initial check for *Remove Empty Sub-Class* refactoring



(b) Execution for *Remove Empty Sub-Class* refactoring

Figure 14. *Remove Empty Sub-Class* refactoring from EMF Refactor.

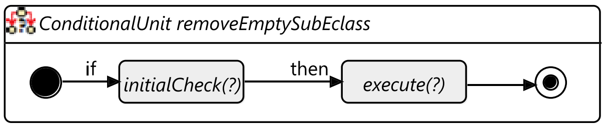


Figure 15. Combination of the initial check and the execution of the *Remove Empty Sub-Class* refactoring.

matching and executing refactorings realized as transformation units. In particular, EMF Refactor uses control-flow units to check whether all pre-conditions of a refactoring are fulfilled before it is actually executed. An example of this is depicted in Figure 14, which demonstrates removal of empty subclasses.

In the initial check, we evaluate whether the selected class is not the super class of another class (*check_superetypes*), whether it is truly empty (*check_empty_eclass*) and whether it is the sub-class of another class (tcheck_subetypes). Only then the actual execution units (cf. Figure 14b) are executed which remove the class from the containing package.

In this case study, we can re-use the most, because EMF Refactor works on Ecore and UML, two meta-models which already have been defined. The refactoring rules are also provided, however, because they are split up into an initial check and an actual execution, we have to first combine them into one transformation unit. Here, we make use of the *Conditional Unit* without the *else*-unit. An example for such a combination is depicted in Figure 15, which uses the initial check as the *if*-unit and the actual execution as the *then*-unit. Please note that, we renamed the rules in Figure 14b because EMF Refactor uses the name ‘mainUnit’ for the combined initial check and the actual execution. Only if the initial check is matched successfully, the actual change will be executed.

The metrics in EMF Refactor can also be re-used by selecting the elements they can be applied upon from the model (`getDomain`) and calculating the respective metric. Listing 4 depicts an example of how we can use the metric calculator that returns the number of all child EClasses of a given EClass by first selecting all EClasses and then calculating the individual values for each class. Re-using available metric calculators from EMF Refactor avoids potential implementation errors and can save time when working with UML or Ecore models.

Listing 4: Using EMF Refactor metric calculator in MOMoT objective

```
SubClasses : minimize {
  val subClassCalculator = new NSUPEC2() // number of all sub-classes
  val eClasses = graph.getDomain(EcorePackage.Literals.ECLASS.eClass, true)
  var subClasses = 0.0
  for(eClass : eClasses) {
    subClassCalculator.context = #[eClass] // eClass as list
    subClasses += subClassCalculator.calculate // reuse EMF Refactor metric
  }
  return subClasses
}
```

Answering RQ1: Summarizing, we have been able to demonstrate the applicability of our approach in different scenarios with different complexity. In this regard, the rules of the different case studies contain varying features in the different examples, and all of them can be integrated in our approach. For instance, the rules in the stack load balancing case study present a prerequisite that is implemented with JavaScript. The class diagram restructuring case study is modeled with complex rules that use the nesting mechanism provided by Henshin, and have also complex NACs, while the use of Henshin units has been exemplified in the EMF Refactor example. Finally, the modularization problem requires the optimization of five objectives in order to obtain proper solutions. This is feasible with the use of many-objective algorithms [25], such as NSGA-III, also included in MOMoT.

RQ2. To evaluate the overhead of our approach, we compare the runtime performance of MOMoT with the performance of a native implementation in the MOEA framework for the modularization and the stack case studies. A native implementation only uses classes from the MOEA framework. In order to provide such an implementation the following steps need to be performed.

1. Choose an appropriate representation (*encoding*) for the decision variables and solution.
2. Map the modularization concepts (*classes*, *modules*, and their *relationships*) to that representation.

3. Provide or select a mechanism to produce new and random solutions.
4. Evaluate the fitness objectives and constraints based on that encoding.
5. Decode the representation into a human-understandable form.

By default, MOEA provides binary variables, grammars, permutations, programs, and real variables as well as a set of selection, crossover, and mutation operators that can work on these variables. For the modularization case study, we use the following strategy. First, all classes are given a unique, consecutive index. Second, we encode the assignments of classes to modules as a sequence of binary variables so that each position in that sequence corresponds to one class and the value of the binary variable indicates the module to which the class is assigned. Therefore, we have as many binary variables as there are classes in the model, for example, for the *mtunis* model (cf. Figure 3) a sequence of 20 binary variables would be used. The length of the binary variable is set to represent at most the number of classes in the model, that is, the case in which all classes are in their own module. Because this is not exactly possible with binary variables, we need to use the next possible value. For instance, to represent 20 modules, we need five bits in a variable enabling an actual range from 0 (00000) to 31 (11111). A representation for the described encoding using the *mtunis* model is depicted in Figure 16.

By choosing binary variables, we can re-use the operators and random solution generators provided by MOEA. However, the evaluation of the fitness of a solution on binary variables is very challenging. We therefore decode the solutions to an internal representation which makes the concepts of classes, modules, and dependencies explicit in order to evaluate the objectives and constraints. This representation can then also be used to present the solutions to the user.

To compare the native encoding with the MOMoT encoding, both approaches are run on the same machine with the same algorithm configuration. The machine is a Lenovo T430S with an Intel Core i5-3320 M CPU 2.60 GHz using 8 GB RAM and running a 64 bit version of Windows 7 Professional. As algorithm, we use the NSGA-III with a maximum of 32768 evaluations, tournament selection with $k=2$, and the one-point crossover operator with $p=1.0$. However, due to the differences in the encoding, we cannot use the exact same mutation operators. In the MOMoT approach, we mutate parts of the solution sequences by substituting them with applicable, random sequences as explained in Section 3.6. In the native approach, we apply a bit-flip mutation which flips each bit of a binary variable, that is, switches a 0 to a 1 and vice versa. Both mutation operators are used with $p=0.1$. To gain insight into the performance, we vary the population size resulting in a different number of iterations, starting with a population size of one and doubling it until we reach the maximum of 32768. Each population size is executed 30 times. A comparison with local search is not possible because MOEA does not provide local search by default.

The results of the NSGA-III comparison are depicted in Figure 17. 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. Table IV depicts the average values and standard deviations for each of the 30 runs. From these experiments, we can observe that the native encoding has a stable runtime performance between 1.5 and 20 s, while the runtime performance of our approach has a bit more variation, because in each execution it varies between 24 and 69 s. Furthermore, our approach performs slower in all execution scenarios.

In a similar manner, we can encode the stack case study where we need to encode the shifting of load parameters between stacks [22]. The stack system (`StackModel` in Figure 10) is represented

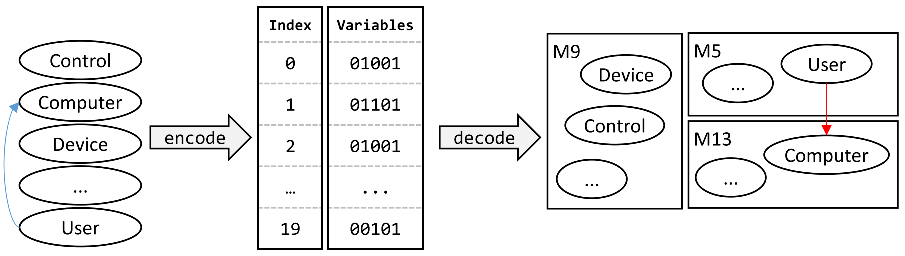


Figure 16. Native encoding for the Modularization case study.

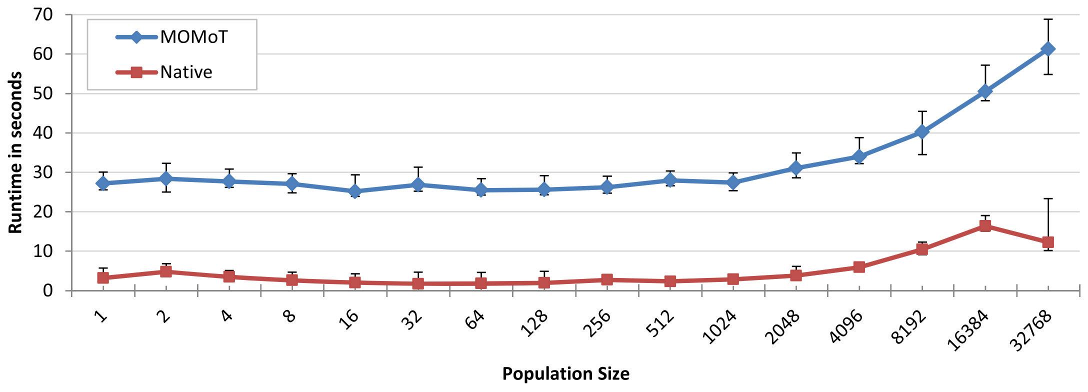


Figure 17. Comparison of total runtime for the Modularization case study: Native encoding vs. MOMoT.

Table IV. Average runtime and standard deviations in milliseconds for the Modularization case study: Native encoding vs. MOMoT.

PopulationSize	MOMoT Avg	Native StdDev	Avg	StdDev
1	27,178	1,141	3,207	833
2	28,394	1,375	4,807	401
4	27,713	846	3,486	317
8	27,082	989	2,658	414
16	25,151	920	2,013	420
32	26,843	1,011	1,753	554
64	25,477	862	1,827	538
128	25,631	968	1,970	570
256	26,249	937	2,734	280
512	27,987	872	2,363	246
1,024	27,396	1,156	2,869	256
2,048	31,065	1,276	3,829	462
4,096	33,967	1,733	5,935	348
8,192	40,233	1,926	10,463	662
16,384	50,472	1,818	16,383	964
32,768	61,282	3,868	12,264	3,220

as a sequence of binary variables. 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. This bit is not considered to be part of the variables integer value. The number of bits to represent the load value is based on the highest load in the provided stack instance, for example, to represent 15, we need four bits (`1111`) and to represent 16, we need five bits (`10000`).

Because we deal with only two objectives, we use the NSGA-II algorithm for the comparison of MOMoT and a native encoding. To execute the experiment, we fix the problem complexity with 100 stacks having a load between 0 and 200 and stop after 10,000 evaluations. To gain insight into the performance, we vary the population size and the number of iterations, respectively. The results of the experiments are depicted in Figure 18 and are similar to what we have seen for the modularization case study. In all cases, our approach performs worse than the opposing native encoding.

This observed loss in performance can be explained twofold. 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

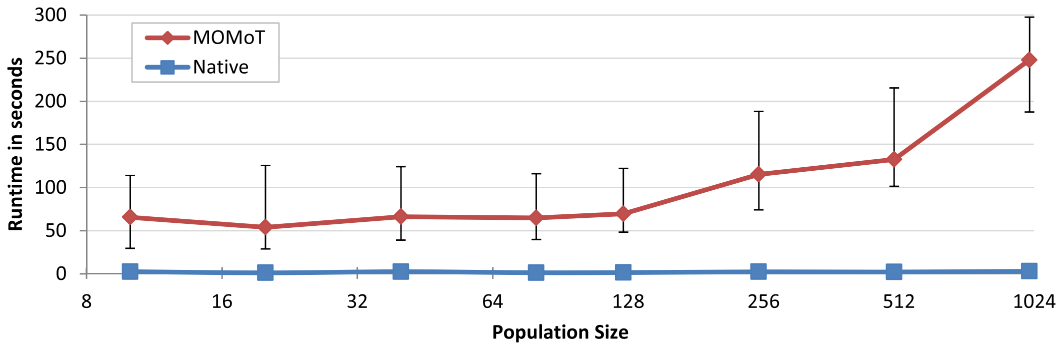


Figure 18. Comparison of total runtime for the Stack Load Balancing case study: Native encoding vs. MOMoT.

a random transformation sequence is more expensive. Second and more importantly, by using graph transformation rules instead of a native encoding, we inherit the complexity of graph pattern matching and match application, which are expensive tasks on their own. In the worst case, the graph pattern matching problem in Henshin is equivalent to the subgraph isomorphism problem [59], which is NP-complete [83].

To further investigate this behavior, we use the JVM Monitor[9] profiler for Java for tracking which operations are the most expensive to execute in MOMoT. The profiling is executed on the same machine as the performance evaluation above and samples the runtime every 50 ms. The results obtained from this profiling show that over 90% of the execution time is spent on finding new matches by evaluating all constraints (NACs, PACs, implicit containment and reference constraints as well as explicit attribute constraints) in the graph and then creating the changes that are applied on that graph. These operations are followed by the graph copy operation, the evaluation of the fitness function and the updating of the population set maintained by the algorithm. The graph copy operation is called whenever a new solution is calculated based on the input graph, that is, the input graph is copied and then the transformation is applied upon that copy resulting in the output graph which is used for the fitness evaluation. The fitness evaluation is called for every created solution and the population set gets updated any time a new solution is added to it, for example, it needs to check whether the solution is dominated by another solution and how the solution relates to the reference points maintained by NSGA-III.

Answering RQ2: Although Henshin uses constraint solving to tackle the problem of finding pattern matches more efficiently, it seems that a certain loss in performance cannot be avoided. This performance loss is also evident when creating a large initial population in the first iteration, where we have to find and apply a lot of random matches to create solutions, which takes a large proportion of the overall execution time, as discussed in [22]. In fact, when profiling the stack case study, where the performance difference between the native encoding and MOMoT was even larger [22], we can see that the larger difference stems from the use of attribute NACs written in JavaScript, which takes a long time to match as it involves the matching of nodes and attributes and the interpretation of JavaScript. Therefore, the performance of MOMoT strongly depends on the complexity of the rules. This has also been observed by Arendt and Taentzer [79] for the EMF Refactor case study, where the time of applying different refactorings can vary from 17 ms (*Inline Class* refactoring) to 236 ms (*Extract Subclass* refactoring).

RQ3. In order to evaluate how our approach contributes and integrates search capabilities to existing MDE frameworks, we compare what search features our approach offers with respect to what is currently available in Henshin. This is performed by investigating what search capabilities are integrated in Henshin and what problems they can solve. Then, we discuss the delta introduced with our approach in order to estimate the effort that is reduced through MOMoT.

[9]JVM Monitor, version 3.8.1, available from http://jvmmonitor.org

In general, with Henshin you need to specify explicitly which transformation unit you want to apply and you need to provide all necessary parameters. The rule engine then tries to find a match for the parameterized transformation unit in the underlying graph. If no matches are found, an error message is displayed. Otherwise, one match is selected either deterministically or non-deterministically, depending on your settings in the rule engine. Multiple units can be applied either manually one after the other or explicitly through a different transformation unit, for instance, the sequential unit. Additionally, Henshin provides the so-called *Henshin State Space Tools* to execute search. These tools are used to verify the correctness of transformations and can generate and analyze a state space given some initial state and the transformation rules. The automatic exploration of Henshin uses a breadth-first search strategy that creates the state space and keeps track of all states that have been visited. Starting from an initial model, all matches for all applicable rules are used to generate a set of new states. Two states are connected through one concrete rule application. In each step of the exploration, a newly created state is checked if it is valid according to a given criteria and if it is a final state (*goal state*) given some other criteria. These criteria can be formulated using OCL or given as a Java class through the configuration. Goal states and states where no transformation rule can be applied are not explored any further (*closed states*). If an invalid state is found, the exploration ends. This is used to evaluate whether the given transformation rules can lead to invalid states by finding counter examples. The automatic explorer stops if all states have been generated and no invalid state has been found. After the exploration process, Henshin provides analysis capabilities to investigate the states space for certain criteria, for example, finding the shortest path from the initial state to some goal state. Summarized, Henshin provides an exhaustive approach to find paths in an explicitly enumerated state space for specified criteria. More complex behavior, such as different search heuristics or an optimization procedure where the concrete criteria cannot be specified in advance, needs to be implemented by the user through the API of Henshin.

MOMoT, on the other hand, does not provide an exhaustive approach, but local search and population-based meta-heuristic approaches. Therefore, without using MOMoT, one can (*i*) use the Henshin rules manually to derive a solution, (*ii*) implement another approach which is able to apply the rules and check the resulting model or (*iii*) adapt the existing State Space Tools. However, due to the complexity of the problems we are dealing with, that is, problems that warrant a meta-heuristic search, a manual approach is clearly not feasible for larger examples. A search-and-check-approach must be able to take back applied transformations in form of backtracking and in most cases more than one possible solution exists, as also observed by Schätz et al. [84]. Therefore, we would at least need to provide rules which are able to negate the effects of another rule. For the modularization case study, this would mean that we need a rule for deleting a class, and another one for removing a feature from a class. For the class diagram restructuring case study, we would need rules to remove root classes and distribute attributes among classes by considering the current class hierarchy in the model. Similar rules need to be implemented for the other case studies. Discarding a manual approach and an approach where we would need to modify the user-provided input, we would aim to adapt the existing functionality in Henshin.

In order to implement local search into Henshin, at least the following aspects need to be implemented:

- The state space explorer must be able to deal with very large or infinite search spaces as they are often created by transformation rules, for example, the rule which creates modules in the modularization case study. Currently, the explorer would traverse the search space until it runs out of memory.
- The evaluation of the quality criteria needs to be implemented and should be executed every time a new state is created. Therefore, a concept of *fitness function* needs to be integrated into the existing state space tools. Invalid states (according to the specified constraints) should be marked and the evaluated quality criteria should be saved with each state.
- Currently the tools only derive new states from rules through matching. An extension which allows the use of other transformation units needs to be developed as otherwise the power of control-flow semantics in units cannot be used.

- Additional search strategies besides breadth-first search need to implemented, for example, depth-first search or A*-search. If possible, the fitness function should guide the direction of the search, similar to what is performed in MOMoT.
- In case multiple quality criteria need to be optimized, the search needs a way to compare different solutions, for example, through the use of the Pareto optimality concept.

In order to implement population-based search, model engineers need to start from scratch. Listing the aspects that need to be considered and implemented in order to provide a similar functionality than MOMoT, it should be clear that this task requires a lot of knowledge and time.

Answering RQ3: Henshin focuses on model-checking and therefore offers an exhaustive search in a bounded, explicitly enumerated search space. However, for the case studies we investigated, such an approach would not yield satisfactory results. First of all, it lacks the capabilities to define optimization goals and second it cannot deal with large search spaces. Furthermore, only transformation rules may be applied, which means that the expressive power of control-flow units is lost. Therefore, a lot of effort needs to be invested in order to solve the given case studies with the default Henshin transformation engine. This was also one of our motivations to propose MOMoT.

Summary. 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 Modularization case study. Furthermore, once a dedicated encoding has been defined, integrating changes may become quite expensive. 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 performed by changing one line in the search configuration. Indeed, this ease of use is especially important for modeling engineers, who are familiar with MDE technologies such as meta-models, 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 to model level, where they can be further processed. The use of dedicated encodings is further complicated by the fact that there is often no common consensus how to solve a specific problem. For instance, whereas the works in [40, 41] 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 introduces an overhead that depends on the complexity of the transformations used in the generic encoding. Nevertheless, our approach enables and guides model engineers to use meta-heuristic search to tackle problems which may not be solvable through an exhaustive approach.

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 model engineers, people from other domains may find these tasks more challenging. Nonetheless, the aim of this approach is to be used in the software engineering process, where models are used as first-class citizens in the design phase, so designers are likely to be knowledgeable about models and model transformations. 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. In any case, to alleviate this threat, we have implemented native encodings using the MOEA framework for both the stack and modularization case studies. These implementations have yielded similar solutions as the ones obtained with our approach.

External validity—To what extent is it possible to generalize the findings? Even though we have selected four 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, 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 performed 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, that is, input/output model pairs. This approach is called model transformation by example (MTBE) [85–87] 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 [88–93]. 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, that is, generating a new model from scratch based on input models, while we focussed in this paper on in-place transformations, that is, rewriting input models to output models. Finally, the authors in [94] 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 [95]. 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 [96], 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.

Searching for model transformation results is currently supported by approaches using some kind of constraint solver. For instance, Kleiner et al. [97] 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. [98] 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 with 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 [99, 100] 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.

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 [19] 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 with 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. [18]. 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 [24] 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 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 (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 [19], integrating the search algorithms into transformation engines [18], or combining transformation engines with search algorithm frameworks as we are doing with MOMoT.

Program transformation is a field closely related to model transformation [101], 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 [102, 103]. 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 RS, 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 [5] as an interesting subject for future work to further evaluate our approach.

6. CONCLUSION AND FUTURE WORK

In MDE, model engineers represent problem domains by means of meta-models, and model transformations are used to manipulate their instances, that is, models. MDE has gained importance in software engineering and beyond. Indeed, models are used throughout the system lifecycle to focus on the important aspects for specific purposes. In order to optimize these models, something that is crucial for the system's success, the models are improved by applying model transformations. Most model transformation approaches are rule-based and require a form of rule orchestration, that is, 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 can be a complex task.

Therefore, in this paper, we have proposed to apply SBSE techniques to support the model engineer 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 model engineers. Besides, we have demonstrated that the use and orchestration of other transformation units, such as control-flow units, is also compatible with our approach, which can reduce the search space and allows us to model more complex problems. Another strong point of our approach is that the user does not need to be an expert in SBSE techniques. In fact, we have implemented an easy-to-use user interface where the user can configure different options for the search process and a static analysis produces warning, errors, and advice when the entered values are not the expected ones.

Although the results presented seem promising, our approach faces some limitations which we plan to address in the future. In fact, when we consider models as graphs and transformation rules and units as graph patterns that must be matched, the problem of finding and applying matches is computationally expensive and reduces the runtime performance. Therefore, first of all, we will further investigate the limitations of our approach with respect to runtime performance. In particular, we foresee working on speeding-up the graph pattern matching by applying parallelization techniques [104] specially in the presence of NACs, as it was identified in Section 4.3 that they worsen the performance, and we will investigate other performance measures such as memory consumption. Second, we have instantiated our approach of merging the MDE and SBSE domains by using Henshin and the MOEA framework, respectively. For this reason, we would like to do experiments with different model transformation approaches and different search engines to see what results we obtain and study if there are improvements in terms of performance, ease of use, or optimization of the solutions. Third, we aim to improve the support of SBSE for novice SBSE users. Therefore, we aim to add additional checks to our support system such as an estimation of the search space based on the input model and the model transformations and we want to evaluate how we can provide additional information about individual solutions to the user. In addition, 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 [105] and to support approximate model transformations [106], that is, trading precision for performance. Finally, we would like to compare our approach with other emerging approaches combining MDE and SBSE such as those presented in [18, 19, 96].

ACKNOWLEDGEMENTS

The authors thank the anonymous reviewers for their insightful and constructive comments and suggestions. This work is funded by the European Commission under FEDER and under ICT Policy Support Programme, grant no. 317859, by the Christian Doppler Forschungsgesellschaft and the BMWFW, Austria, and by the Spanish and the Andalusian R&D&I programmes under grants and projects BELI (TIN2015-70560-R), THEOS (P10-TIC-5960) and COPAS (P12-TIC-1867).

REFERENCES

1. Brambilla M, Cabot J, Wimmer M. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers: London, UK, 2012. DOI:10.2200/S00441ED1V01Y201208SWE001.
2. Sendall S, Kozaczynski W. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software* 2003; **20**(5):42–45. DOI:10.1109/MS.2003.1231150.
3. Law AM, Kelton DM. *Simulation Modeling and Analysis*, 3rd edn. McGraw-Hill Higher Education: Columbus, Ohio, USA, 1999.

4. Smith JS. Survey on the Use of Simulation for Manufacturing System Design and Operation. *Journal of Manufacturing Systems* 2003; **22**(2):157–171. DOI:10.1016/S0278-6125(03)90013-6.
5. Bruneliere H, Cabot J, Jouault F, Madiot F. MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. *Proceedings of the 25th International Conference on Automated Software Engineering* (*ASE'10*), IEEE/ACM, 2010; 173–174. DOI:10.1145/1858996.1859032.
6. Bergmayr A, Grossniklaus M, Wimmer M, Kappel G. JUMP-From Java Annotations to UML Profiles. *Proceedings of 17th International Conference on Model-Driven Engineering Languages and Systems* (*MoDELS'14*), *LNCS*, vol. 8767. Springer, 2014; 552–568. DOI:10.1007/978-3-319-11653-2_34.
7. Czarnecki K, Helsen S. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal* 2006; **45**(3):621–645. DOI:10.1147/sj.453.0621.
8. Lúcio L, Amrani M, Dingel J, Lambers L, Salay R, Selim G, Syriani E, Wimmer M. Model Transformation Intents and Their Properties. *Software and System Modeling* 2014; 1–38. DOI:10.1007/s10270-014-0429-x.
9. Kurtev I. State of the Art of QVT: A Model Transformation Language Standard. *Proceedings of the 3rd International Symposium on Applications of Graph Transformations with Industrial Relevance* (*AGTIVE'07*), *LNCS*, vol. 5088, Springer, 2008; 377–393. DOI:10.1007/978-3-540-89020-1_26.
10. Schürr A. Specification of Graph Translators with Triple Graph Grammars. *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science* (*WG'94*), *LNCS*, vol. 903, Springer, 1995; 151–163. DOI:10.1007/3-540-59071-4_45.
11. Rivera JE, Duran F, Vallecillo A. A Graphical Approach for Modeling Time-dependent Behavior of DSLs. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (*VL/HCC'09*), IEEE, 2009; 51–55. DOI:10.1109/VLHCC.2009.5295300.
12. Taentzer G. AGG: A Graph Transformation Environment for Modeling and Validation of Software. *Proceedings of the 2nd Workshop on Applications of Graph Transformations with Industrial Relevance* (*AGTIVE'03*), *LNCS*, vol. 3062. Springer, 2003; 446–453. DOI:10.1007/978-3-540-25959-6_35.
13. Agrawal A. Graph Rewriting And Transformation (GReAT): A Solution For The Model Integrated Computing (MIC) Bottleneck. *Proceedings of the 18th International Conference on Automated Software Engineering* (*ASE'03*), IEEE, 2003; 364–368. DOI:10.1109/ASE.2003.1240339.
14. Jouault F, Allilaire F, Bézivin J, Kurtev I. ATL: A model transformation tool. *Science of Computer Programming* 2008; **72**(1–2):31–39. DOI:10.1016/j.scico.2007.08.002.
15. Csertán G, Huszerl G, Majzik I, Pap Z, Patariczа A, Varró D. VIATRA - visual automated transformations for formal verification and validation of UML models. *Proceedings of the 17th International Conference on Automated Software Engineering* (*ASE'02*), IEEE/ACM, 2002; 267–270. DOI:10.1109/ASE.2002.1115027.
16. Arendt T, Biermann E, Jurack S, Krause C, Taentzer G. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems* (*MoDELS'10*), *LNCS*, vol. 6394, Springer, 2010; 121–135. DOI:10.1007/978-3-642-16145-2_9.
17. Born K, Arendt T, Hess F, Taentzer G. Analyzing Conflicts and Dependencies of Rule-Based Transformations in Henshin. *Proceedings of the International Conference on Fundamental Approaches to Software Engineering* (*FASE'15*), *LNCS*, vol. 9033. Springer, 2015; 165–168. DOI:10.1007/978-3-662-46675-9_11.
18. Abdeen H, Varró D, Sahraoui HA, Nagy AS, Debreceni C, Hegedüs Á, Horváth Á. Multi-objective optimization in rule-based design space exploration. *Proceedings of the 29th International Conference on Automated Software Engineering* (*ASE'14*), IEEE/ACM, 2014; 289–300. DOI:10.1145/2642937.2643005.
19. Denil J, Jukss M, Verbrugge C, Vangheluwe H. Search-Based Model Optimization Using Model Transformations. *Proceedings of the 8th International Conference on System Analysis and Modeling* (*SAM'14*), *LNCS*, vol. 8769, Springer, 2014; 80–95. DOI:10.1007/978-3-319-11743-0_6.
20. Kessentini M, Langer P, Wimmer M. Searching models, modeling search: On the synergies of SBSE and MDE. *Proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering* (*CMSBSE'13*) *@ ICSE*, IEEE, 2013; 51–54. DOI:10.1109/CMSBSE.2013.6604438.
21. Harman M. The Current State and Future of Search Based Software Engineering. *Proceedings of the Workshop on the Future of Software Engineering* (*FOSE'07*) *@ ICSE*, IEEE Computer Society, 2007; 342–357. DOI:10.1145/1253532.1254729.
22. Fleck M, Troya J, Wimmer M. Marrying Search-based Optimization and Model Transformation Technology. *Proceedings of the 1st North American Search Based Software Engineering Symposium* (*NasBASE'15*), 2015.
23. Kurtev I, Aksit M, Bézivin J. Technological Spaces: An Initial Appraisal. *Proceedings of Federated Conferences CoopIS*, *DOA*, *and ODBASE*, 2002.
24. Deb K, Pratap A, Agarwal S, Meyarivan T. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 2002; **6**(2):182–197. DOI:10.1109/4235.996017.
25. 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 Transactions on Evolutionary Computation* 8 2014; **18**(44):577–601. DOI:10.1109/TEVC.2013.2281535.
26. Deb K, Mohan M, Mishra S. A Fast Multi-objective Evolutionary Algorithm for Finding Well-Spread Pareto-Optimal Solutions. *Technical Report 2003002*, Indian Institute of Technology Kanpur 2003.
27. Zitzler E, Thiele L, Laumanns M, Fonseca CM, da Fonseca VG. Performance Assessment of Multiobjective Optimizers: An Analysis and Review. *IEEE Transactions on Evolutionary Computation* 2003; **7**(2):117–132. DOI:10.1109/TEVC.2003.810758.

28. Holland JH. *Adaptation in Natural and Artificial Systems*. MIT Press: Cambridge, Mass., USA, 1992.
29. Glover F. Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers and Operations Research* 1986; **13**(5):533–549. DOI:10.1016/0305-0548(86)90048-1.
30. Kirkpatrick S, Gelatt CD, Vecchi MP. Optimization by Simulated Annealing. *Science* 1983; **220**(4598):671–680.
31. Kühne T. Matters of (Meta-)Modeling. *Software and System Modeling* 2006; **5**(4):369–385. DOI:10.1007/s10270-006-0017-9.
32. Object Management Group (OMG). *Meta Object Facility* (*MOF*) *2.0 Core Specification* 2003. OMG Document ptc/03-10-04.
33. Object Management Group (OMG). *Object Constraint Language* (*OCL*) *Specification*. *Version 2.2* 2010. OMG Document formal/2010-02-01.
34. Mens T, Gorp PV. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science* 2006; **152**:125–142. DOI:10.1016/j.entcs.2005.10.021.
35. Chen Z, Liu Z, Ravn AP, Stolz V, Zhan N. Refinement and verification in component-based model-driven design. *Science of Computer Programming* 2009; **74**(4):168–196. DOI:10.1016/j.scico.2008.08.003.
36. de Lara J, Vangheluwe H. AToM3: A Tool for Multi-formalism and Meta-modelling. *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering* (*FASE'02*), *LNCS*, vol. 2306. Springer, 2002; 174–188. DOI:10.1007/3-540-45923-5_12.
37. Syriani E, Vangheluwe H, Mannadiar R, Hansen C, Mierlo SV, Ergin H. AToMPM: A Web-based Modeling Environment. *Joint Proceedings of Invited Talks*, *Demonstration Session*, *Poster Session*, *and ACM Student Research Competition @ MoDELS'13*, CEUR-WS, 2013; 21–25.
38. Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N, Meseguer J, Talcott C. *All About Maude – A High-Performance Logical Framework*, *LNCS*, vol. 4350. Springer, 2007. DOI:10.1007/978-3-540-71999-1.
39. Praditwong K, Harman M, Yao X. Software Module Clustering as a Multi-Objective Search Problem. *IEEE Transactions on Software Engineering* 2011; **37**(2):264–282. DOI:10.1109/TSE.2010.26.
40. Seng O, Stammel J, Burkhart D. Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems. *Proceedings of the 8th Conference on Genetic and Evolutionary Computation* (*GECCO*), 2006; 1909–1916. DOI:10.1145/1143997.1144315.
41. Harman M, Tratt L. Pareto Optimal Search Based Refactoring at the Design Level. *Proceedings of the Genetic and Evolutionary Computation Conference* (*GECCO'07*), ACM, 2007; 1106–1113. DOI:10.1145/1276958.1277176.
42. Masoud H, Jalili S. A clustering-based model for class responsibility assignment problem in object-oriented analysis. *Journal of Systems and Software* 2014; **93**(0):110–131. DOI:10.1016/j.jss.2014.02.053.
43. Holt RC. *Concurrent Euclid, the Unix* System, and Tunis*. Addison-Wesley: Reading, Mass., 1983.
44. Ó Cinnéide M, Tratt L, Harman M, Counsell S, Hemati Moghadam I. Experimental Assessment of Software Metrics Using Automated Refactoring. *Proceedings of the International Symposium on Empirical Software Engineering and Measurement* (*ESEM'12*), ACM: New York, NY, USA, 2012; 49–58. DOI:10.1145/2372251.2372260.
45. Abdeen H, Ducasse S, Sahraoui H. Modularization metrics: Assessing package organization in legacy large object-oriented software. *Proceedings of the 18th Working Conference on Reverse Engineering* (*WCRE'11*), IEEE Computer Society, 2011; 394–398. DOI:10.1109/WCRE.2011.55.
46. Yourdon E, Constantine LL. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, 1st edn. Prentice-Hall: New Jersey, USA, 1979.
47. de Oliveira Barros M. Evaluating Modularization Quality as an Extra Objective in Multiobjective Software Module Clustering. *Proceedings of the 3rd International Symposium on Search Based Software Engineering* (*SSBSE'11*), *LNCS*, vol. 6956. Springer: Berlin, Heidelberg, 2011; 267–267. DOI:10.1007/978-3-642-23716-4_23.
48. Goldberg DE, Lingle Jr R. Alleles, loci, and the traveling salesman problem. *Proceedings of the 1st International Conference on Genetic Algorithms* (*ICGA'85*), Lawrence Erlbaum Associates, 1985; 154–159.
49. Harman M, McMinn P, Teixeira de Souza J, Yoo S. Search Based Software Engineering: Techniques, Taxonomy, Tutorial. *Proceedings of the International Summer School on Empirical Software Engineering and Verification* (*LASER'10*), *LNCS*, vol. 7007, Springer, 2010; 1–59. DOI:10.1007/978-3-642-25231-00_1.
50. Mann HB, Whitney DR. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics* 1947; **18**(1):50–60. DOI:10.1214/aoms/1177730491.
51. Kruskal WH, Wallis WA. Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association* 1952; **47**(260):583–621. DOI:10.2307/2280779.
52. Sheskin DJ. *Handbook of Parametric and Nonparametric Statistical Procedures*, 3 edn. Chapman & Hall/CRC: Washington, 2003.
53. Cohen J. *Statistical Power Analysis for the Behavioral Sciences*, 2 edn. Lawrence Erlbaum Associates: New Jersey, USA, 1988.
54. Vargha A, Delaney HD. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Education and Behavioral Statistics* 2000; **25**(2):101–132. DOI:10.3102/10769986025002101.
55. Cliff N. Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions. *Psychological Bulletin* 1993; **114**(3):494–509. DOI:10.1037/0033-2909.114.3.494.
56. Romano J, Kromrey JD, Coraggio J, Skowronek J. Appropriate Statistics for Ordinal Level Data: Should We Really Be Using t-test and Cohen's d for Evaluating Group Differences on the NSSE and other Surveys? 2006. Proceedings of the Annual Meeting of the Florida Association of Institutional Research.
57. Fleck M, *Troya J*, Wimmer M. MOMoT Project 2015. Available at http://martin-fleck.github.io/momot.

58. Biermann E, Ermel C, Taentzer G. Lifting Parallel Graph Transformation Concepts of Model Transformation based on the Eclipse Modeling Framework. *Electronic Communications of the EASST* 2010; **26**:1–19.
59. Tichy M, Krause C, Liebel G. Detecting Performance Bad Smells for Henshin Model Transformations. *Proceedings of the 2nd Workshop on the Analysis of Model Transformations* (*AMT'13*) *@ MoDELS*, vol. 1077, CEUR-WS, 2013.
60. Mancoridis S, Mitchell BS, Rorres C, Chen Y, Gansner ER. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. *Proceedings of the 6th International Workshop on Program Comprehension* (*IWPC'98*), 1998; 45–52. DOI:10.1109/WPC.1998.693283.
61. Mitchell BS, Mancoridis S. On the Automatic Modularization of Software Systems Using the Bunch Tool. *IEEE Transactions on Software Engineering* 2006; **32**(3):193–208. DOI:10.1109/TSE.2006.31.
62. Mitchell BS, Mancoridis S. On the Evaluation of the Bunch Search-Based Software Modularization Algorithm. *Soft Computing* 2008; **12**(1):77–93. DOI:10.1007/s00500-007-0218-3.
63. Harman M, Hierons RM, Proctor M. A New Representation And Crossover Operator For Search-based Optimization Of Software Modularization. *Proceedings of the 4th Conference on Genetic and Evolutionary Computation* (*GECCO'02*), Morgan Kaufmann, 2002; 1351–1358.
64. Abdeen H, Ducasse S, Sahraoui HA, Alloui I. Automatic Package Coupling and Cycle Minimization. *Proceedings of the 16th Working Conference on Reverse Engineering* (*WCRE'09*), IEEE, 2009; 103–112.
65. Mkaouer W, Kessentini M, Shaout A, Koligheu P, Bechikh S, Deb K, Ouni A. Many-Objective Software Remodularization Using NSGA-III. *ACM Transactions on Software Engineering and Methodology* 2015; **24**(3):17:1–17:45. DOI:10.1145/2729974.
66. Fowler M. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.
67. Qayum F, Heckel R. Search-Based Refactoring using Unfolding of Graph Transformation Systems. *Electronic Communications of the EASST* 2011; **38**:1–14.
68. Mens T, Taentzer G, Runge O. Analysing refactoring dependencies using graph transformation. *Software and System Modeling* 2007; **6**(3):269–285. DOI:10.1007/s10270-006-0044-6.
69. Harman M, Jones BF. Search-based software engineering. *Information & Software Technology* 2001; **43**(14):833–839. DOI:10.1016/S0950-5849(01)00189-6.
70. O'Keeffe MK, Ó Cinnéide M. Search-based refactoring for software maintenance. *Journal of Systems and Software* 2008; **81**(4):502–516.
71. Mkaouer MW, Kessentini M, Bechikh S, Deb K, Ó Cinnéide M. High dimensional search-based software engineering: Finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III. *Proceedings of the Genetic and Evolutionary Computation Conference* (*GECCO'14*), ACM, 2014; 1263–1270. DOI:10.1145/2576768.2598366.
72. O'Keeffe M, Ó Cinnéide M. A Stochastic Approach to Automated Design Improvement. *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java* (*PPPJ'03*), Computer Science Press, 2003; 59–62. DOI:10.1145/957289.957308.
73. Gorp PV, Rose LM, Krause C (eds.). *Proceedings of the 6th Transformation Tool Contest* (*TTC'13*), *EPTCS*, vol. 135, Open Publishing Association, 2013. DOI:10.4204/EPTCS.135.
74. Lano K, Kolahdouz Rahimi S. Case study: Class diagram restructuring. *Proceedings of the 6th Transformation Tool Contest* (*TTC'13*), *EPTCS*, vol. 135, Open Publishing Association, 2013; 8–15. DOI:10.4204/EPTCS.135.2.
75. Kolahdouz-Rahimi S, Lano K, Pillay S, Troya J, Gorp PV. Evaluation of model transformation approaches for model refactoring. *Science of Computer Programming* 2014; **85Part A**(0):5–40. DOI:10.1016/j.scico.2013.07.013.
76. Chidamber SR, Kemerer CF. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 1994; **20**(6):476–493. DOI:10.1109/32.295895.
77. Genero M, Piattini M, Calero C. *Metrics For Software Conceptual Models*. Imperial College Press, 2005.
78. Eclipse. EMF Refactor 2015. Available at http://www.eclipse.org/emf-refactor.
79. Arendt T, Taentzer G. A tool environment for quality assurance based on the Eclipse Modeling Framework. *Automated Software Engineering* 2013; **20**(2):141–184. DOI:10.1007/s10515-012-0114-7.
80. Bechikh S, Ben Said L, Ghédira K. Searching for Knee Regions of the Pareto Front Using Mobile Reference Points. *Soft Computing* 2011; **15**(9):1807–1823. DOI:10.1007/s00500-011-0694-3.
81. Godin R, Mili H. Building and Maintaining Analysis-level Class Hierarchies Using Galois Lattices. *Proceedings of the 8th Annual Conference on Object-oriented Programming Systems, Languages, and Applications* (*OOPSLA'93*), ACM, 1993; 394–410. DOI:10.1145/165854.165931.
82. Berry A, Gutierrez A, Huchard M, Napoli A, Sigayret A. Hermes: a simple and efficient algorithm for building the AOC-poset of a binary relation. *Annals of Mathematics and Artificial Intelligence* 2014; **72**(1-2):45–71. DOI:10.1007/s10472-014-9418-6.
83. Cook SA. The complexity of theorem-proving procedures. *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing* (*STOC'71*), ACM, 1971; 151–158. DOI:10.1145/800157.805047.
84. Schätz B, Hölzl F, Lundkvist T. Design-Space Exploration through Constraint-Based Model-Transformation. *Proceedings of the 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems* (*ECBS'10*), IEEE, 2010; 173–182. DOI:10.1109/ECBS.2010.25.
85. Varró D. Model Transformation by Example. *Proceedings of 9th International Conference on Model-Driven Engineering Languages and Systems* (*MoDELS'06*), *LNCS*, vol. 4199, Springer, 2006; 410–424. DOI:10.1007/11880240_29.

86. Wimmer M, Strommer M, Kargl H, Kramler G. Towards Model Transformation Generation By-Example. *Proceedings of the 40th Hawaii International International Conference on Systems Science* (*HICSS'07*), IEEE, 2007; 285b. DOI:10.1109/HICSS.2007.572.
87. Kappel G, Langer P, Retschitzegger W, Schwinger W, Wimmer M. Model Transformation By-Example: A Survey of the First Wave. *Conceptual Modelling and Its Theoretical Foundations, LNCS* 2012; **7260**, Springer: 197–215. DOI:10.1007/978-3-642-28279-9_15.
88. Kessentini M, Sahraoui HA, Boukadoum M. Model Transformation as an Optimization Problem. *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems* (*MoDELS'08*), *LNCS*, vol. 5301, Springer, 2008; 159–173. DOI:10.1007/978-3-540-87875-9_12.
89. Kessentini M, Bouchoucha A, Sahraoui HA, Boukadoum M. Example-Based Sequence Diagrams to Colored Petri Nets Transformation Using Heuristic Search. *Proceedings of the 6th European Conference on Modelling Foundations and Applications* (*ECMFA'10*), *LNCS*, vol. 6138, Springer, 2010; 156–172. DOI:10.1007/978-3-642-13595-8_14.
90. Kessentini M, Sahraoui HA, Boukadoum M, Benomar O. Search-based model transformation by example. *Software and System Modeling* 2012; **11**(2):209–226. DOI:10.1007/s10270-010-0175-7.
91. Baki I, Sahraoui HA, Cobbaert Q, Masson P, Faunes M. Learning Implicit and Explicit Control in Model Transformations by Example. *Proceedings of 17th International Conference on Model-Driven Engineering Languages and Systems* (*MoDELS'14*), *LNCS*, vol. 8767, Springer, 2014; 636–652. DOI:10.1007/978-3-319-11653-2_39.
92. Faunes M, Sahraoui HA, Boukadoum M. Genetic-Programming Approach to Learn Model Transformation Rules from Examples. *Proceedings of the 6th International Conference on Theory and Practice of Model Transformations* (*ICMT'13*), *LNCS*, vol. 7909, Springer, 2013; 17–32. DOI:10.1007/978-3-642-38883-5_2.
93. Saada H, Huchard M, Nebut C, Sahraoui HA. Recovering model transformation traces using multi-objective optimization. *Proceedings of the 28th International Conference on Automated Software Engineering* (*ASE'13*), IEEE/ACM, 2013; 688–693. DOI:10.1109/ASE.2013.6693134.
94. Shelburg J, Kessentini M, Tauritz D. Regression Testing for Model Transformations: A Multi-objective Approach. *Proceedings of the 5th International Symposium on Search Based Software Engineering* (*SSBSE'13*), *LNCS*, vol. 8084. Springer, 2013; 209–223. DOI:10.1007/978-3-642-39742-4_16.
95. ben Fadhel A, Kessentini M, Langer P, Wimmer M. Search-based detection of high-level model changes. *Proceedings of the 28th IEEE International Conference on Software Maintenance* (*ICSM'12*), IEEE, 2012; 212–221. DOI:10.1109/ICSM.2012.6405274.
96. Efstathiou D, Williams JR, Zschaler S. Crepe complete: Multi-objective optimisation for your models. *Proceedings of the First International Workshop on Combining Modelling with Search- and Example-Based Approaches* (*CMSEBA'14*) *@ MODELS*, vol. 1340, CEUR-WS.org, 2014; 25–34.
97. Kleiner M, Didonet Del Fabro M, Queiroz Santos D. Transformation as search. *Proceedings of the 9th European Conference on Modelling Foundations and Applications* (*ECMFA'13*), *LNCS*, vol. 7949, Springer, 2013; 54–69. DOI:10.1007/978-3-642-39013-5_5.
98. Gogolla M, Hamann L, Hilken F. On static and dynamic analysis of UML and OCL transformation models. *Proceedings of the Workshop on Analysis of Model Transformations* (*AMT*) *@ MODELS*, *CEUR Workshop Proceedings*, vol. 1277, CEUR-WS.org, 2014; 24–33.
99. Drago ML, Ghezzi C, Mirandola R. Towards Quality Driven Exploration of Model Transformation Spaces. *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems* (*MoDELS'11*), *LNCS*, vol. 6981, Springer, 2011; 2–16. DOI:10.1007/978-3-642-24485-8_2.
100. Drago ML, Ghezzi C, Mirandola R. A quality driven extension to the QVT-relations transformation language. *Computer Science - R&D* 2015; **30**(1):1–20. DOI:10.1007/s00450-011-0202-0.
101. Visser E. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science* 2001; **57**(2):109–143. DOI:10.1016/j.jsc.2004.12.011.
102. Fatiregun D, Harman M, Hierons RM. Search Based Transformations. *Proceedings of the Genetic and Evolutionary Computation Conference* (*GECCO'03*), *LNCS*, vol. 2724, Springer, 2003; 2511–2512. DOI:10.1007/3-540-45110-2_154.
103. Fatiregun D, Harman M, Hierons RM. Evolving transformation sequences using genetic algorithms. *Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation* (*SCAM'04*), IEEE, 2004; 66–75. DOI:10.1109/SCAM.2004.11.
104. Burgueño L, Troya J, Wimmer M, Vallecillo A. On the Concurrent Execution of Model Transformations with Linda. *Proceedings of the Workshop on Scalability in Model Driven Engineering* (*BigMDE*) *@ STAF*, ACM, 2013; 3:1–3:10. DOI:10.1145/2487766.2487770.
105. Fleck M, Troya J, Langer P, Wimmer M. Towards Pattern-Based Optimization of Cloud Applications. *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud* (*CloudMDE*) *@ MoDELS*, vol. 1242, CEUR-WS.org, 2014; 16–25.
106. Troya J, Wimmer M, Burgueño L, Vallecillo A. Towards Approximate Model Transformations. *Proceedings of the Workshop on Analysis of Model Transformations* (*AMT*) *@ MoDELS*, vol. 1277, CEUR-WS.org, 2014; 44–53.