Model-Driven Optimization: Towards Performance-Enhancing Low-Level Encodings

Lars van Arragon
Radboud University Nijmegen (NL)

Carlos Diego Damasceno
Radboud University Nijmegen (NL)

Daniel Strüber
Chalmers | University of Gothenburg (SE), Radboud University Nijmegen (NL)

***Abstract*—In Model-Driven Optimisation, meta-heuristic optimization algorithms are applied to models to solve optimization problems. A meta-model is used to describe a modelling language which defines the search space. Exploration operators (e.g., mutation) are usually expressed as model transformations. During the search space exploration, transformations as well as model copying can become a performance bottleneck, significantly slowing down performance. In this paper, as a first step towards solving this issue, we contribute a low-level encoding of models that does not replace, but compliments them. The encoding stores information about the mutable parts of the model in a way that is inexpensive to change and copy, whereas other operations (e.g., querying of non-mutable parts) are still performed on the actual model. We include a formal framework for expressing what such an encoding looks like, together with an implementation on top of MDEOptimiser, an existing tool for Model-Driven Optimization. In a performance evaluation on two scenarios, we find improved performance in one, and new, clearly identified performance challenges in a second scenario.**

***Index Terms*—modeling techniques, optimization**

I. Introduction

Optimization problems—problems of finding a best or near-best solution among multiple feasible ones—play an important role in our everyday lives and manifold professional contexts, such as healthcare, education, and finance. Search-Based Software Engineering (SBSE, [1]) describes the application of meta-heuristic techniques to optimization problems in software engineering. Examples of problems for which the use has been widely explored in a SBSE context are test case selection, design space exploration, and requirements prioritization.

Model-Driven Optimisation (MDO, [2]) combines principles from Model-Driven Engineering [3] and SBSE to increase the abstraction level for specifying optimization problems and their solution using meta-heuristic search techniques. In MDO, users specify the search space for multi-objective optimization problems in the form of a meta-model with associated constraints and fitness functions. Each valid model represents one solution candidate. Exploration operators such as mutation are implemented as model transformations that query and update solution candidate models. This provides several key benefits, including improved usability as users interact with declarative domain models, and potential performance improvements, as one can define domain-specific mutation [4]–[6] and crossover operators [7], [8] that avoid constraint violations or have other useful properties. There exist several tools that apply this concept, including MDEOptimiser [9], MoMOT [10], VIATRA-DSE [11], and FitnessStudio [12].

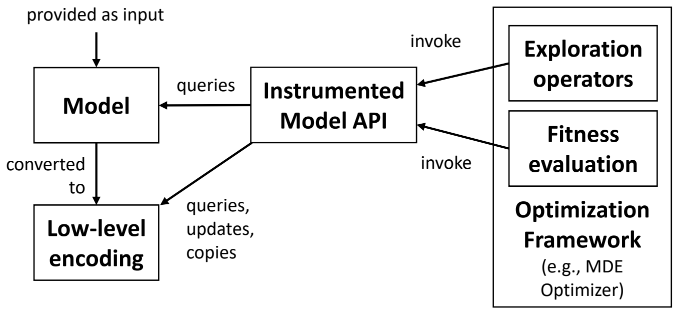

Fig. 1: Overview of approach

Despite its user-oriented benefits and performance-oriented motivation, MDO can be affected by performance drawbacks, especially when applied to large problem instances. Executing a meta-heuristic optimisation algorithm on models requires that a large population of solution candidates is maintained and evolved through model transformations, which leads to many model querying and updating operations based on an available Model API (e.g., an automatically generated one from EMF [13]). Such Model APIs are often not optimized for performance in this use-case, in which many versions of the same model (solution candidates) exist in parallel in one population. Furthermore, models are routinely copied to create new variants for the mutation and crossover steps. Since these models are big data structures with a lot of information, their copying can become a computational bottleneck.

Our long-term vision is an approach that enables all computational steps to be performed on low-level encodings (i.e., based on primitive types and standard collections), while the user specifies the problem and interacts exclusively with declarative domain models and model transformations. Such a solution is promising to yield significant improvements, offering the best of both worlds. However, it presents an ambitious technical challenge, as it entails questions such as: how to execute exploration operators defined as model transformations directly on low-level encodings?

In this paper, we present the first steps towards improving the performance of MDO approaches by using low-level encodings as the basis upon which the optimisation algorithms are applied. We provide an approach in which solution candidates are expressed as low-level encodings that exist in parallel to the model. Some of the invocations of model queries – those that involve mutable parts of solutions – are performed on the

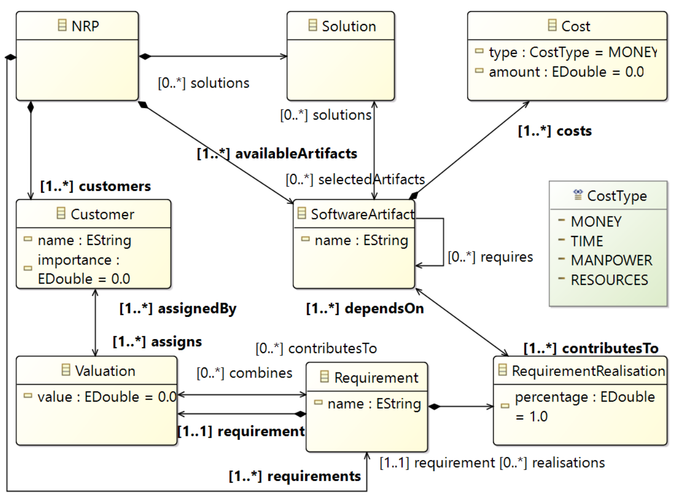

Fig. 2: Meta model for NRP [9]

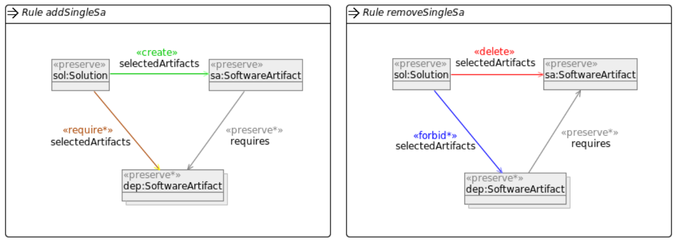
Rule addSingleSa
«preserve» sol:Solution
«create» selectedArtifacts
«preserve» sa:SoftwareArtifact
«require*» selectedArtifacts
«preserve*» requires
«preserve*» dep:SoftwareArtifact
Rule removeSingleSa
«preserve» sol:Solution
«delete» selectedArtifacts
«preserve» sa:SoftwareArtifact
«forbid*» selectedArtifacts
«preserve*» requires
«preserve*» dep:SoftwareArtifact

Fig. 3: Mutation rules for NRP [9]

low-level encoding, while others are performed on the model itself. The overview in Fig. 1 shows the components of our approach: a formally defined low-level encoding which we can automatically generate from the input model, as well as instrumented model API that ensures that queries and updates to the mutable model parts from exploration operators and fitness evaluation are performed on the low-level encoding. In particular, this approach reduces the effort associated with the copying of models.

Specifically, we make the following contributions:

- We present a formal framework for expressing what an encoding looks like for any given model instance and meta-model.
- Based on this framework, we implemented a Java library for encoding any meta-model with model instance that is expressed within the EMF framework.
- We discuss and show how the instrumented model API can be implemented for two particular MDO scenarios.
- We conduct several experiments in the two scenarios to validate the potential performance improvements.
- We discuss the results, providing pointers for future work.

For our implementation, we focus on the MDEOptimiser tool, since it provides out-of-the-box support for the considered scenarios, and has shown favorable performance in an experimental comparison in a previous study [14]. However, our solution is generally applicable for EMF-based frameworks for model-driven optimization. All of the code written for this paper can be found on GitHub[1][2].

II. Background and Related Work

MDO by Example. We now illustrate the concept of MDO introduced in Sect. I on an example, the Next Release Problem (NRP, [15]). The essence of NRP is that we have several software artifacts that have a cost of realisation and an importance for a customer. These customers also have a differing importance. Now the question becomes what artifacts do we realize for the next release? We want to minimise the cost of the realisation and maximize the customer satisfaction.

In an MDO framework, this problem can be modeled using the meta-model shown in Fig. 2. Notably, the meta-model has a small part that is mutable in solution candidates, i.e., the edge between *Solution* and *SoftwareArtifact* denoting which artifacts are contained in the current solution, whereas most other parts remain constant over solution candidates (e.g., the cost of particular artifacts). In addition to the meta-model, the problem definition includes constraints (e.g., C_1*: if an artifact is included in the solution, all of its dependencies need to be included as well*), and fitness functions (e.g., F_1*: user satisfaction is to be maximized*). In MDEOptimizer [9], constraints such as C_1 can be specified in OCL, Xtend or Java.

MDO further involves the definition of problem-specific exploration operators, typically using model transformation languages. Such operators can be either manually specified or automatically generated. Fig. 3 shows mutation operators for assigning and un-assigning artifacts to and from solutions, respectively, specified as transformation rules in Henshin [16]. The (un-)assignment is achieved via the *create* and *delete* edges. Both rules are designed to ensure that a particular constraint – C1 as defined above – is respected during mutations. In particular, the *requires** edge in addSingleSa denotes an application condition, ensuring that artifacts are only added to solutions if all dependent artifacts are in the solution as well.

Previous encoding approaches. Previous work on combining model-driven and search-based software engineering has explored the use of generic encodings of MDE models [17], [18]. However, during exploration, they relied on the standard mutation and cross-over operators, whereas our approach aims to make domain-specific exploration operators defined as model transformation applicable to the encoding. The NRP example illustrates a benefit of such exploration operators: given a bit-vector encoding of NRP, applying the standard ”bit-flip” mutation operator may lead to a constraint violation regarding constraint C1. While search-based optimization algorithms are geared to deal with invalid solutions (in particular, during selection), producing and then later discarding invalid solutions might not be the most efficient strategy. Previous work [5] shows a benefit of using tailored mutation operators that never produce invalid solutions over existing approaches.

Domain-specific search operators. An inherent benefit of our approach is its potential to support domain-specific search

[1]https://github.com/larsvanarragon/mde_optimiser-hilo/tree/nobitset
[2]https://github.com/larsvanarragon/mde_optimiser/tree/encoding

operators specified using model transformations, as opposed to generic search operators, which have been used in previous encoding-based approaches. A recent research line addresses the specification of such search operators, either in the context of a specific domain (e.g., product line configuration [5] or feature allocation [19]–[21]) or by providing a generic operator generation approach that addresses constraint-aware encodings [4] or uses a notion of meta-learning to generate optimized domain-specific operators [12]. Follow-up work has addresses the analysis of search operators to ensure that they either remove or at least do not create new constraint violations [22], and to specify search operators in a flexible way, to have optional changes in search operators that are only executed if required by a constraint [23].

Constraint satisfaction problems. In the context of design space exploration, there exists related work on declarative DSLs with an automated transformation to a low-level encoding in the context of constraint satisfaction problems (CSP). Saxena et al. [24], [25] have proposed an approach in which users use a high-level DSL and constraint language for problem specification, which they transform to an intermediate, solver independent encoding supporting several back-end CSP solvers. Chenouard et al. [26] have proposed an approach for topology optimization, in which the user models components and requirements, and automated transformations to and from the MiniZinc CSP solver are provided. While low-level CSP solvers such as MiniZinc offer some support for optimization problems, they address a different scope than our scope of search-based optimization, in particular: support is limited to mono-objective problems and exact optimization (identifying global maxima), which is desirable, but for large problem instances terminates without result.

III. MDO using a Low-Level Encoding

A. Encoding

We now introduce our encoding in the form of a formal definition. We aim to support a way to flexibly encode model instances in which both model elements and references between model elements can be added and removed. To that end, we assume a graph-based representation of the meta-model (C, R) and a model instance (V, E), with C being the set of classes, R being the set of references, V being the set of objects (vertices), and E being the set of reference instances (edges), respectively. The meta-model and model we consider here contain exactly the information that needs to be contained in the encoding, that is, classes and references which can be subject to additions and deletions. They might be subgraphs of a larger graphs that represent the full context of the optimization problem (e.g., the full NRP case and an instance).

We define a mapping from a meta model and a model instance to an encoding $\mathcal{P}(R_A, \mathcal{P}(R_I, \mathcal{P}(L)))$. Here we have that R_A is a set of labels that denote the abstract relations within the meta-model, R_I is a set of labels that denote the relation instances within the model instance and $\mathcal{P}(L)$ is the powerset of labels. The powersets in the encoding denote that we have several abstract relationships who in turn have several relationship instances. All specific label sets like R_A and R_I have that $R_I \subseteq L$ and $R_A \subseteq L$. The usage of these names is just for convenience.

To enable the encoding to link between label representations of the object instances we assume that there is a bijective function **id** : $V \to L$ from which we can obtain an identifier for any object instance $v \in V$. With its corresponding inverse which we call **obj** : $L \to V$ which links any identifier to its object. Lastly we also assume that we have an operator $+$ which concatenates any two $k, l \in L$ labels. To start of this definition we need to gather all of the referenced objects of any object within the model instance as identifiers and relate them to the identifier of the object that references them.

Definition 1 (Relation instance)**.** *Suppose a graph (V, E) with a vertex $v \in V$, a set of relation labels R_I and a set of labels L. We call $r_I : (R_I, \mathcal{P}(L))$ a **relation instance** of v if $r_I = (id(v), \{\ id(w') \mid (w, w') \in E \wedge w = v\ \})$*
*Suppose a graph (V, E). We define **RelInst**(V, E) as all relation instances for (V, E). Namely, we define **RelInst**$(V, E) = \{\ r_I \mid v \in V \wedge r_I$ is a **relation instance** of $v\}$.*

This gives us a set containing all relation instances for the graph. We now want to obtain all relation instances for any given abstract relation. For this we restrict the relation instances to a relation in the corresponding meta-model.

Definition 2 (Restricted set of relation instances)**.** *Suppose a meta-model (C, R) and a corresponding model instance (V, E). For an arbitrary $r \in R$ we define **RelInstRestr**(V, E, r) as all of the relation instances for (V, E) restricted to r. Namely, we define **RelInstRestr**$(V, E, r) = \{\ (v, W/U) \mid (v, W) \in RelInst(V, E) \wedge U = \{\ w \mid w \in W \wedge (obj(v), obj(w)) \in E \wedge (obj(v), obj(w)) \notin r\} \wedge W/U \neq \emptyset\}$.*

In essence, this definition removes all of the edges from *RelInst* that are not an instance of the meta relation r. Using this restriction we can now define what an encoded abstract relation looks like.

Definition 3 (Encoded relationship)**.** *Suppose a graph (V, E) which is a model instance of a meta-model (C, R) with a relation $r = ((c, P), (d, Q), l, Q) \in R$. We call r_A : $(R_A,$ **RelInstRestr**$(V, E, r))$ an **encoded relationship** of r if $r_A = (l + id(c) + id(d), \{\ r_I \mid r_I \in RelInstRestr(V, E, r)\})$*

We can now construct the formal encoding for any given meta-model (C, R) with a corresponding model instance (V, E). For every $r \in R$ we can construct an *encoded relationship* and take a set of all of them.

Definition 4 (Encoding)**.** *Suppose a graph (V, E) which is a model instance of a meta-model (C, R). We define an **encoding** of (C, R) and (V, E) to be $\{\ r_A \mid r \in R \wedge r_A$ is an encoded relationship of $r\}$*

Example. We apply the above definitions to the small NRP example shown in Fig. 4. Considering the relevant meta-model exerpt as (C, R) and the instance (V, E), we obtain:

$$\begin{aligned}
C &= \{(Solution, S), (Artifact, A)\} \\
R &= \{((Solution, S), (Artifact, A), selectedArtifacts, SEL), \\
&\quad ((Artifact, A), (Solution, S), solutions, SOL), \\
&\quad ((Artifact, A), (Artifact, A), requires, REQ)\} \\
V &= \{proposedSolution, mainArtifact, \\
&\quad drawArtifact, logArtifact\} \\
E &= \{(proposedSolution, mainArtifact), \\
&\quad (mainArtifact, proposedSolution), \\
&\quad (mainArtifact, drawArtifact), \\
&\quad mainArtifact, logArtifact)\}
\end{aligned} \tag{1}$$

First, we look at what a relation instance looks like for some vertex. This small example only has two relation instances, as there are only two classes that have outgoing relations. We show them for mainArtifact and proposedSolution:

$$\begin{aligned}
r_{I_{mainArtifact}} &= (id(mainArtifact), \\
&\quad \{id(proposedSolution), id(drawArtifact), \\
&\quad id(logArtifact)\}) \\
r_{I_{proposedSolution}} &= (id(proposedSolution), \\
&\quad \{id(mainArtifact)\})
\end{aligned} \tag{2}$$

Taking both $r_{I_{mainArtifact}}$ and $r_{I_{proposedSolution}}$ as a set now already is **RelInst**(V, E). We can now look at what all restricted relations look like for (V, E), but we will focus only on those of mainArtifact as here we can restrict to different abstract relations. We can either restrict to $r_0 = ((Artifact, A), (Solution, S), solutions, SOL)$ which gives:

$$\begin{aligned}
\textbf{RelInstRestr}(V, E, r_0) = \{(id(mainArtifact), \\
\{id(proposedSolution)\})\}
\end{aligned} \tag{3}$$

Alternatively, we can restrict to $r_1 = ((Artifact, A), Artifact, A), requires, REQ)$, yielding:

$$\begin{aligned}
\textbf{RelInstRestr}(V, E, r_1) = \{(id(mainArtifact), \\
\{id(drawArtifact), id(logArtifact)\})\}
\end{aligned} \tag{4}$$

This enables us to encode one of these restricted relationship instances. Let us pick r_0, we have then that r_A is an **encoded relationship** for r_0 if:

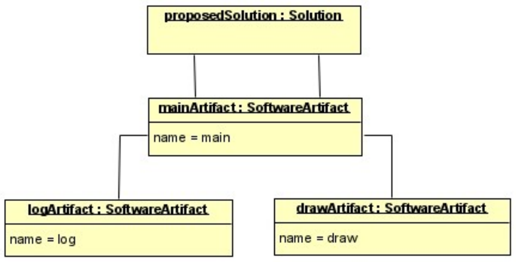
proposedSolution : Solution
mainArtifact : SoftwareArtifact
name = main
logArtifact : SoftwareArtifact
name = log
drawArtifact : SoftwareArtifact
name = draw

Fig. 4: Simple NRP instance

$$\begin{aligned}
r_A = (&solutions + id(Artifact, A) + \\
&id(Solution, S), \{(id(mainArtifact), \\
&\{id(proposedSolution)\})\})
\end{aligned} \tag{5}$$

The entire **encoding** of (C, R) and (V, E) can now be instantiated, leading to the set shown in Fig. 5.

B. Embedding in Model-Driven Optimisation

Using the above definitions, we have encoded a meta-model with its model instance. However, this is not yet enough to actually use the encoding within any given MDO technique. To enable this, we have to consider two things. One, how do we evolve the encoding instead of the model? Two, how do we evaluate the encoding with the fitness function? Both of these questions concern other, user-defined aspects of the optimization setup, specifically, the fitness function and evolution operators. We do not want a user of an MDO technique to consider the encoding. That would defeat the entire purpose of using models as a way of representing the population. Our goal is to make sure the user of such an MDO technique does not even realise it is being encoded.

One way to enable the seamless integration of the encoding is to intercept relevant function calls to the model and supply the information based on the encoding. To do this it is important that we add an extra ingredient to the encoding. A bidirectional map of the **id** : $V \rightarrow L$ and **obj** : $L \rightarrow V$. We call this bidirectional map the 'Repository', and it implements the storage of all relevant model instances. To use the Repository we fill it with the initial objects in the model instance during the encoding phase. Then, whenever an exploration operator adds an object instance to the model, we instantiate the object, generate an identifier for it, and add it to the Repository. If an exploration operator removes an object from the model, the Repository should remove the relevant mapping to save space. Note that maintaining the bidirectional map increases the required memory for storing information for our encoding; at the same time, it does so to a smaller extent than our considered baseline of storing solution candidates as modified copies of the same model in some standard model representation (e.g., EMF).

With the addition of this Repository it is now easy to intercept functions on the model. From the context of the function we can retrieve the relevant abstract relationship. We can then, based on the object on which it is called, retrieve the relevant identifiers it is related to within the encoding, and return the list of actual objects back to caller. We will illustrate this idea using the small NRP model example—recall Fig. 2. A logical function every Solution has is to get its selected artifacts. The context gives us the identifier for this encoded relationship. Namely, $selectedArtifacts + id(Solution, S) + id(Artifact, A)$. With this we can then obtain all relationship instances for this abstract relationship. Now suppose we call this function on the proposedSolution. We can then invoke $id(proposedSolution)$ to lookup the relevant set of identifiers that this Solution references. We then simply return a new

$$\begin{aligned}&\{(solutions + id(Artifact, A) + id(Solution, S), \{(id(mainArtifact), \{id(proposedSolution)\})\}),\\&(requires + id(Artifact, A) + id(Artifact, A),\\&\qquad\{(id(mainArtifact), \{id(drawArtifact), id(logArtifact)\})\}),\\&(selectedArtifacts + id(Solution, S) + id(Artifact, A), \{(id(proposedSolution), \{id(mainArtifact)\})\})\}\end{aligned}$$

Fig. 5: Complete encoding

set that contains the actual object instances for the set of identifiers.

This way both the fitness functions and exploration operators can still treat the population as models, while they are actually using the encoding. Ideally, the generation of this interception should be done automatically. This was out of scope for the paper and will thus be left as future work.

C. Implementation

We implemented all of the aforementioned concepts as a Java library for the EMF framework. The library currently supports the automatic generation of the encoding and the automatic lookup of related objects. Within the library there are three important classes that facilitate all the functionality for the encoding to be applied in any situation. These are the classes Converter, Encoding, and Repository, as we can see in the class diagram in Fig. 6. The ModelLoader class is utility for loading in the files.

The converter exposes an interface that accepts any meta-model ecore file and a model instance xmi file and converts it to an encoding. The Repository class is implemented as described in the previous section, it contains the bidirectional map between object instances and identifiers. It supplies functionality for the id and obj functions. The Encoding class itself contains as data a HashMap that has as keys all of the abstract relationships within the meta-model and as values another HashMap which contains the **RelInstRestr** for that abstract relationship.

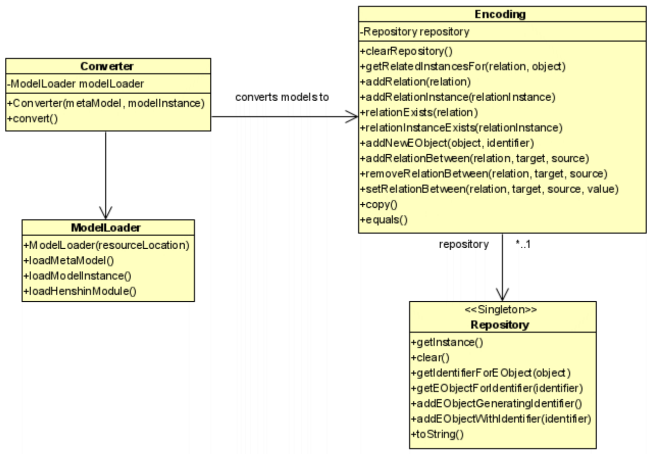
Converter
-ModelLoader modelLoader
+Converter(metaModel, modelInstance)
+convert()
converts models to
Encoding
-Repository repository
+clearRepository()
+getRelatedInstancesFor(relation, object)
+addRelation(relation)
+addRelationInstance(relationInstance)
+relationExists(relation)
+relationInstanceExists(relationInstance)
+addNewEObject(object, identifier)
+addRelationBetween(relation, target, source)
+removeRelationBetween(relation, target, source)
+setRelationBetween(relation, target, source, value)
+copy()
+equals()
ModelLoader
+ModelLoader(resourceLocation)
+loadMetaModel()
+loadModelInstance()
+loadHenshinModule()
repository
*..1
<<Singleton>>
Repository
+getInstance()
+clear()
+getIdentifierForEObject(object)
+getEObjectForIdentifier(identifier)
+addEObjectGeneratingIdentifier()
+addEObjectWithIdentifier(identifier)
+toString()

Fig. 6: Encoding Java library class model

A user of this library should only ever have to work with the Converter.convert() function and the lookup function from the Encoding class. This gives a two layered approach where the model exists within the code to guide the user to not construct violations of the models within their implementation. The encoding should now be used to do the heavy lifting of the optimisation algorithm, namely the evolution and copying.

Within the library we also support the addition of new objects to the repository as described within the previous section. However, we do not support the removal of objects from the repository, as efficient garbage collection is a big challenge in itself, out of the scope of this paper.

The interception of the calls on the models (*Instrumented Model API* in Fig. 1) is not part of the Java library. We discuss two options to implement the instrumentation, before explaining our choice taken for our implementation and experiments. The first is a wrapper that is generated with the generation of the actual code for the models by EMF. The wrapper then functions as the bridge between the model and the encoding and should as such be context aware, which is possible because it is generated based on the model. The second is to intercept the calls to the models using aspect-oriented programming, e.g., in its Java implementation AspectJ [27]. Since AspectJ does not require us to interfere with the code generation of EMF and gives us relative easy access to the interfaces we need this is what we chose to implement the interception of the calls within the experiments we performed.

IV. Evaluation

To validate our concepts and implementation, we consider two scenarios for which an available solution in our baseline tool MDEOptimiser was available: NRP [15] and CRA [28]. We apply our encoding and its implementation on top of MDEOptimizer to both scenarios. We measured the time for completion of the optimisation based on the population size and the number of evaluations. All experiments were executed on a Windows 11 machine (CPU: Intel i7 7700K 4 cores; RAM: 16GB DDR4-3200 CL16) with Java 11.

A. Set-up

This section presents the set-up for the experiments done for NRP and CRA, including main design decisions.

Next Release Problem. Our first scenario is NRP. From the meta-model for NRP from Fig. 2, we can observe that the crucial relation for this problem lies between Solution and SoftwareArtifact. The core of the NRP is to decide for each software artifact whether it is added to the solution or not. Hence, the obtained encoding in this case precisely captures this information, rendering it essentially a bit vector. The rest of this model will always stay the same for each member of the population.

We use the Java implementation of the NRP provided in [9], on which we implemented the MOEA framework. As we need to evaluate two variants we implemented two problems that represent them. The encoded problem was quite straightforward as we could simply use the provided BinaryVariable given by MOEA. For the model problem however, we had to implement our own custom variable.

In this experiment, we used the standard operators for evolving a bit vector provided by the MOEA framework, namely the HUX and BitFlip operators. The HUX operator, also called the half uniform crossover operator, is an operator which accepts two parents and then compares their vectors. For parents p and q with vectors $p_1, \cdots, p_n$ and $q_1, \cdots, q_n$ we have that if $p_i \neq q_i$ then their values are swapped. In a usual case there is only a chance that this swap happens, but in the case of this experiment this chance is 100%. The BitFlip operator accepts one parent and has for each bit a 1% chance to flip its value to the opposite. In this experiment, we apply HUX first and then apply BitFlip on the resulting children. We let the MOEA framework handle the operators for the encoded variant of the problem, and we implement these operators for the model variant ourselves by instantiating a Variation class. This implementation should behave exactly the same as the standard implementations from MOEA.

To intercept the calls to the model we use AspectJ to redirect them to obtain the information from our encoding. This is quite simple to implement, as the only calls that we have to intercept are the getters in the SoftwareArtifact and Solution classes that reference each other.

We used two available model instances for this experiment, differing in their size. The smaller one, called model A, included 5 customers, 25 requirements, and 63 software artifacts. The bigger one, called model B, included 25 customers, 50 requirements, and 203 software artifacts. For each model instance we ran the experiment on two variables; namely, the population size and amount of evaluations. Per combination of these variables we run the experiment several times. Our goal was to identify whether the configuration of these variables has an impact on the observed performance differences.

Class Responsibility Assignment. Our implementation of CRA is based on the available implementation provided by [9], augmented with our generated encoding. To ensure that our encoding instead of the model is used for queries to mutable parts, we used AspectJ again to intercept the relevant calls to the model and redirect them to the encoding.

We made every effort to stay as close as possible to the available MDEOptimiser solution, to ensure that any observed performance differences come from our encoding. We copied the way MDEOptimiser calls MOEA, and then filled in our custom AbstractProblem, Variable and Variation that are to be used during the optimisation algorithm. We used the exploration operators as provided by [9]. Like MDEOptimiser, we rely on Henshin for the pattern matching of the rule to the model. To enable this we implemented a stub for the graph that Henshin expects that is filled with the information of the encoding. We then tell Henshin to pattern match to that graph, which calls the relevant getters in the model. Henshin then gives us back a list of changes that should happen to the model, which we apply to the encoding instead.

To measure the performance we recorded the time it took for each of the actions as described in the methodology to complete as well as the overall time. Since these steps get executed multiple times throughout the algorithm we take the average of each of these times per experiment. To measure it for MDEOptimiser we add measurements as hard coded pieces within the corresponding classes in MDEOptimiser.

We use five model instances within this experiment provided by [9]. These model instances range from 9 features with 14 dependencies to 160 features and 600 dependencies. For the population size we have chosen 40 with 500 evaluations per member of the population (i.e. $20,000$ total evaluations). For this case, to obtain detailed information about the different components of the total time taken, we measure the total time for the experiments to complete and the average time per experiment it took to complete: copying a solution candidate, applying mutation, and evaluating a solution candidate.

B. Results

In this section we will show and discuss the results gotten from the experiments we conducted. Firstly, we will discuss the results for the NRP case study. Then we will move on to the results for the CRA case study.

NRP. Table I gives an overview of the results obtained for the NRP case. Within NRP we performed the experiment on two problem instances (models). While we performed the results with varying parametrizations for population size and number of evaluations, we only show the results for varying numbers of evaluations, as the trend we observed only shows linear growth when higher iteration numbers are considered. Population size had no effect on the performance, since the amount of calculations is solely determined by the number of iterations. A higher population size just resulted in every member of the population receiving a smaller portion of the number of iterations. We observe that the search was generally twice as fast when our encoding was used, indicating a clear benefit of the encoding over the baseline.

Following standard recommendations for evaluations of meta-heuristic algorithms [29], we tested our results for statistical significance, using t-testing in the available implementation *scipy.stats.ttest*. For both models and all considered comparisons, the obtained p-values range between e^{-9} and e^{-26}, which is greatly below the predefined threshold of $\alpha = 0.05$. In consequence, we can conclude that the observed differences regarding execution time are statistically significant.

As a correctness check, we considered the quality of the produced solutions for both treatments, with the encoding and

TABLE I: Results for NRP case (times in sec.), with number of evaluations (E), median times and standard deviations (SD)

Model	E	Encoding		Baseline	
		Median time	**SD**	**Median time**	**SD**
A	10K	4.5	0.16	8.8	0.31
	20K	8.9	0.17	17.6	0.44
	30K	13.0	0.14	25.8	0.35
B	10K	14.7	0.66	28.3	0.57
	30K	42.6	0.14	83.7	0.33
	50K	70.6	0.08	138.5	0.28

in the baseline, measured in terms of the hypervolume indicator, a standard quality indicator in multi-objective problems, present in the original case [9]. Our implementation is correct if there is no significant difference between the quality of its solutions and the solutions produced by the baseline. For all experiments, we calculated the hypervolumes for each variant with respect to a previously calculated pareto front of the model instance. This pareto front is the combination of the result of running both experiments with a large population size and amount of evaluations. To validate significance, we again applied t-testing. In t-test, a statistically significant difference is observed if the obtained p-value is greater than the predefined significance threshold. The obtained p-value was greater than 0.05 in all cases, indicating no significant difference and, therefore, a correct implementation.

CRA. Table II gives an overview for the results obtained for the CRA case. For the smaller cases A and B, the overall results look promising for the encoding: according to the *total runtime* column, the encoding experiments completed on average faster for the model instances A and B. For model instance C the encoding experiments where on average as fast as MDEOptimiser, but had greater variance among the values. For the larger model instances D and E we see the opposite picture: the encoding is much slower. To further understand a possible cause for these results we measured the time it took an experiment to complete several steps.

We started with looking at the average time it took to copy members of the population. The results for these experiments are shown in the *copying time* column in the table. From these results we can observe that, like in the NRP case study, the encoding beats out MDEOptimiser for every model instance. Having an average speedup of 2.36 for the biggest model instance E. This means that we have to look at the other steps within the algorithm that interact with the encoding.

We first consider the average time taken to mutate a member of the population. These results are shown in the *mutation time* column. We observe the same trend as in the total runtime column, we start of faster in models A and B, we have similar results for model C and are slower for the bigger models.

We move on to the evaluation of the members of the population. The results for this step are visible in the *evaluation time* column. These results immediately show that the encoding is slower when evaluating a member of the population. This slowdown becomes even more noticeable when the size of the model increases, which is the inverse of what we aimed to achieve. Similar to the observations for the mutation time, the time taken for large models increases by a factor between two and three.

We again performed tests for statistical significance and correctness using the same strategy as for NRP, based on t-testing and the problem-specific quality indicator called CRA index. We found that the observed performance differences for all models except for model C are statistically significant, with p-values between e^{-5} and e^{-35}. The absence of significance for model C, with $p = 0.44$, is expected, as C, in terms of model size, is exactly at the turning point from when the baseline approach performs favorably. For the correctness test, again we found no significant difference between the obtained solutions produced with our encoding compared to the baseline, indicating correctness of our implementation.

In conclusion, considering CRA, the fitness evaluation and mutation of members of the population is slower in the experiments that implement the encoding than in MDEOptimiser itself. Whereas the copying of the members of the population is still faster when using the encoding. We will further discuss these results in the next section.

V. Discussion

This section will discuss the results of our experiments. In the case of the NRP, we see that the results show that in both cases of the model instance using an encoding as the representation is a lot quicker than using the model (Table I). However, for CRA, we observe slowdowns for the two largest models. When looking at the different components of the overall execution times, we see that both mutation and evaluation became slower when our encoding was used (Table II). The worst offender in the CRA case is Henshin's pattern matching during mutation. Now there can be several reasons for this observation. One is that since we use AspectJ to intercept all calls to the model, the overhead that is created due to this makes it impossible to have faster or even comparable times. This makes the most amount of sense in our opinion, AspectJ is known to increase overhead somewhat [30], but that is usually deemed negligible due to the amount of calls typical Java programs have to libraries, which dwarfs the amount of time spent in user code. In our case however, we spent a lot of time in the user code, as we capture any call towards the domain model. It is also the case that a simple getter that returns part of the model is faster than having to gather the right objects from the object repository, as this list needs to be constructed every time. When combining this with the fact that these getters are called a lot, we can easily see how just a slight decrease in performance can lead to seconds or even minutes when running an entire optimisation algorithm. Another explanation could be that using an encoding this way and intercepting calls to the models just is not viable as it costs more in performance than it gains.

Another aspect noticed during the implementation of the encoding is that if we have a bit vector as the underlying data structure, lookups through this vector would take at worst n

TABLE II: Results for CRA case, with median times and standard deviations (SD)

	Total runtime (in seconds)				**Copying time (in microseconds)**				**Mutation time (in microseconds)**				**Evaluation time (in microseconds)**			
	Encoding		**Baseline**		**Encoding**		**Baseline**		**Encoding**		**Baseline**		**Encoding**		**Baseline**	
Model	**t med.**	**SD**	**t med.**	**SD**	**t med.**	**SD**	**t med.**	**SD**	**t med.**	**SD**	**t med.**	**SD**	**t med.**	**SD**	**t med.**	**SD**
A	1.6	0.3	2.3	0.3	4.4	0.5	13.4	1.4	45.4	9.9	67.0	10.8	25.5	5.4	18.5	3.0
B	3.8	1.2	4.5	1.0	9.2	1.4	24.9	3.8	105.3	42.8	127.2	33.4	71.2	16.7	55.4	9.7
C	16.0	5.3	15.3	4.2	17.0	1.1	42.8	2.2	541.9	231.6	519.1	188.9	238.0	36.2	182.2	21.5
D	97.9	26.0	64.0	13.8	23.1	0.9	53.6	2.6	4134.8	1233.9	2808.4	669.0	740.0	71.3	318.5	22.8
E	843.8	126.3	376.9	50.9	53.7	1.20	114.9	4.3	39207.6	6157.3	17403.3	2487.0	3004.4	183.6	1273.1	57.7

amount of cycles where n is the amount of objects within a class. This was because in the beginning we did not take into account that Henshin rules can add and remove objects from the model. To alleviate this issue we adapted the encoding to use a set of identifiers instead, which cut the amount of cycles to only those object that are actually related to the object.

The ideas we represent within this paper can be extrapolated to tools outside of MDEOptimiser. We think that the core concept, the formal encoding, is a useful way of encoding any meta-model with model instances. Which can solve, as we show, the performance of copying the population.

The main problem that we see now is that MDEOptimiser and the surrounding tools are not tailored towards using such an encoding. We thought that using AspectJ would not impact the performance as much as it did. However, we can not yet be certain that it is the fault of AspectJ that the performance is worse. It could just be the case that this way of encoding the models only benefits the copying of the population. More research is needed to figure whether we can get MDEOptimiser to a place where the overall performance of the encoded variant is as fast or faster than the model variant.

A further aspect that could potentially affect performance is our choice to store encoded information in HashMaps of strings. Arguably, HashMaps are a reasonable choice for our prototypical implementation, due to their low cost for lookups in $O(1)$. However, the use of dedicated model indexers such as *Hawk* [31] could lead to performance improvements.

Another big issue is the usability of the encoding. In an ideal setting we do not want the end user that describes the domain and exploration operators to perfectly tune their models for the encoding. This is to preserve the advantages that a tool like MDEOptimiser introduces. In the CRA experiment we enabled this with the use of AspectJ, as that enabled us to redirect calls to the model to the encoding. While we, for this prototypical implementation, implemented the AspectJ integration manually, it could, in principle, be automatically generated. An orthogonal direction for future work, as outlined in the beginning of this paper, is to provide advanced tools that can perform fitness evaluations and mutations directly on the encoding, instead of using an intermediate layer such as the instrumented model API. Such tools would contribute to the potential to keep the overall optimization framework as a black-box to the end-user.

VI. Conclusion

We presented a formal low-level encoding for solution candidates in the context of model-driven optimization, together with an implementation based on AspectJ for instrumenting the used model API to forward relevant queries to the encoding, and experiments from two scenarios from the literature.

From our experiments, we can draw several conclusions. The first is that our encoding can generally lead to improved performance, in terms of execution time, as evidenced by the results from the NRP case. While the NRP case is conceptually simple, it is representative for a wider class of problems known as knapsack problems, in which items are selected for inclusion into some container, as to maximize value. However, as we see from the results from the CRA case, the way we implemented the encoding within MDEOptimiser using AspectJ does not increase the overall performance. This implies that the current implementation does not generally improve performance in MDO applications. On a more positive note, our results confirm that copying solution instances was much faster in both experiments.

We foresee several directions for future work. The most crucial direction is to enable apply the application of evolution rules directly on the encoding, instead of using AspectJ to mediate. Another interesting aspect to explore is related to the overhead AspectJ creates. It could be worth altering the way the model code is generated from the models the user creates, if this takes the encoding into account, it is possible to diminish the negative impact of AspectJ even more and maybe even completely stop using it to intercept calls to the model. We also intend to provide support for problems in which attributes are altered by exploration operators, and apply our approach to a broader selection of cases.

References

[1] M. Harman, P. McMinn, J. T. de Souza, and S. Yoo, *Search Based Software Engineering: Techniques, Taxonomy, Tutorial*, pp. 1–59. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[2] S. Zschaler and L. Mandow, "Towards model-based optimisation: Using domain knowledge explicitly," in *STAF Workshops 2016. Revised selected papers*, pp. 317–329, Springer, 2016.

[3] M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2017.

[4] A. Burdusel, S. Zschaler, and S. John, "Automatic generation of atomic consistency preserving search operators for search-based model engineering," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 106–116, IEEE, 2019.

[5] J. M. Horcas, D. Strüber, A. Burdusel, J. Martinez, and S. Zschaler, "We're not gonna break it! consistency-preserving operators for efficient product line configuration," *TSE'23: IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1102–1117, 2023.

[6] N. van Harten, C. D. N. Damasceno, and D. Strüber, "Model-driven optimization: Generating smart mutation operators for multi-objective problems," in *SEAA'22: Euromicro Conference Series on Software Engineering and Advanced Applications*, pp. 390–397, 2022.

[7] S. John, J. Kosiol, and G. Taentzer, “Towards a configurable crossover operator for model-driven optimization,” in *MDEIntelligence: Workshop on Artificial Intelligence and Model-Driven Engineering at MODELS*, pp. 388–395, 2022.

[8] H. Thölke and J. Kosiol, “A multiplicity-preserving crossover operator on graphs,” in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pp. 588–597, 2022.

[9] A. Burdusel, S. Zschaler, and D. Strüber, “MDEoptimiser: A search based model engineering tool,” in *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS ’18, (New York, NY, USA), p. 12–16, Association for Computing Machinery, 2018.

[10] M. Fleck, J. Troya, and M. Wimmer, “Marrying search-based optimization and model transformation technology,” *Proc. of NasBASE*, pp. 1–16, 2015.

[11] H. Abdeen, D. Varró, H. Sahraoui, A. S. Nagy, C. Debreceni, Á. Hegedüs, and Á. Horváth, “Multi-objective optimization in rule-based design space exploration,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 289–300, 2014.

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

[13] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[14] S. John, A. Burdusel, R. Bill, D. Strüber, G. Taentzer, S. Zschaler, and M. Wimmer, “Searching for optimal models: Comparing two encoding approaches,” *Journal of Object Technology*, vol. 18, pp. 6:1–22, July 2019. The 12th International Conference on Model Transformations.

[15] A. J. Bagnall, V. J. Rayward-Smith, and I. M. Whittley, “The next release problem,” *Information and software technology*, vol. 43, no. 14, pp. 883–890, 2001.

[16] D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, and M. Tichy, “Henshin: A usability-focused framework for EMF model transformation development,” in *International Conference on Graph Transformation*, pp. 196–208, Springer, 2017.

[17] M. Kessentini, P. Langer, and M. Wimmer, “Searching models, modeling search: On the synergies of SBSE and MDE,” in *2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, pp. 51–54, IEEE, 2013.

[18] D. Efstathiou, J. R. Williams, and S. Zschaler, “Crepe complete: Multi-objective optimization for your models.,” in *CMSEBA@ MoDELS*, pp. 25–34, 2014.

[19] K. Born, S. Schulz, D. Strüber, and S. John, “Solving the class responsibility assignment case with Henshin and a genetic algorithm,” in *TTC@ STAF*, pp. 45–54, 2016.

[20] A. Burdusel and S. Zschaler, “Model optimisation for feature class allocation using MDEOptimiser: A TTC 2016 submission.,” in *TTC@ STAF*, pp. 33–38, 2016.

[21] A. S. Nagy and G. Szárnyas, “Class responsibility assignment case: a Viatra-DSE solution,” pp. 39–44, 2016.

[22] J. Kosiol, D. Strüber, G. Taentzer, and S. Zschaler, “Sustaining and improving graduated graph consistency: A static analysis of graph transformations,” *SCP’22: Science of Computer Programming*, vol. 214, pp. 102729–1, 2022.

[23] J. Kosiol, D. Strüber, G. Taentzer, and S. Zschaler, “Finding the right way to Rome: Effect-oriented graph transformation,” in *ICGT’23: International Conference on Graph Transformations*, 2023.

[24] T. Saxena and G. Karsai, “Towards a generic design space exploration framework,” in *2010 10th IEEE International Conference on Computer and Information Technology*, pp. 1940–1947, IEEE, 2010.

[25] T. Saxena and G. Karsai, “A meta-framework for design space exploration,” in *2011 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, pp. 71–80, IEEE, 2011.

[26] R. Chenouard, C. Hartmann, A. Bernard, and E. Mermoz, “Computational design synthesis using model-driven engineering and constraint programming,” in *STAF Workshops 2016. Revised selected papers*, pp. 265–273, Springer, 2016.

[27] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, “Getting started with AspectJ,” *Communications of the ACM*, vol. 44, no. 10, pp. 59–65, 2001.

[28] M. Fleck, J. Troya Castilla, and M. Wimmer, “The class responsibility assignment case,” *Transformation Tool Contest*, pp. 1–8, 2016.

[29] A. Arcuri and L. Briand, “A practical guide for using statistical tests to assess randomized algorithms in software engineering,” in *Proceedings of the 33rd international conference on software engineering*, pp. 1–10, 2011.

[30] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge, “Measuring the dynamic behaviour of AspectJ programs,” *SIGPLAN Not.*, vol. 39, p. 150–169, oct 2004.

[31] K. Barmpis and D. Kolovos, “Hawk: Towards a scalable model indexing architecture,” in *Proceedings of the workshop on scalability in model driven engineering*, pp. 1–9, 2013.