# Model Optimisation for Feature–Class allocation using MDEOptimiser: A TTC 2016 Submission

Alexandru Burdusel
Steffen Zschaler
Department of Informatics, King's College London
szschaler@acm.org

May 13, 2016

**Abstract**

In this paper, we present a solution for the TTC 2016 challenge case on class–responsibility assignment. Our solution is built on our early prototype of MDEOptimiser, a tool integrating meta-heuristic search into model-driven engineering.

## 1 Introduction

Search-based software engineering (SBSE) is about using optimisation techniques for automating the search for (near-)optimal software designs [3]. Using search-based techniques allows the exploration of much larger design spaces than could be explored manually by developers. As a result, better solutions can be identified more quickly. Model-driven engineering (MDE) offers unique benefits to SBSE because it already comes with good techniques for expressing design spaces (aka meta-models) and for deriving new solution candidates from existing ones (aka model transformations). There is, then, a need to support the expression and execution of optimisation tasks in the context of MDE. The current TTC case study (asking to find an optimal assignment of interdependent features to classes as part of an object-oriented design process) is an excellent challenge example for demonstrating such capabilities.

In this paper, we show how we have tackled this problem with our MDEOptimiser tool. Our tool is still very young, so while we can solve the problem and explain our solution in this paper, we place substantial focus on the lessons learned from this exercise and how these will inform future development of the tool.

The remainder of this paper is structured as follows: We begin with a brief overview of the challenge case and our tool. Section 4, then, presents our solution

to the challenge case, which we evaluate in Sect. 5 based on the criteria defined with the case. Section 6 discusses lessons learned from this experiment and highlights future research to be undertaken. Section 7 concludes the paper.

## 2 Case Study Description

This submission addresses the TTC "Class Responsibility Assignment Case" provided by Fleck *et al.*.[1] In this section, we give a brief overview of this challenge case to provide context for the remainder of the paper. A more detailed description of the case study is available in the full case description on GitHub.

The challenge case is about a key step in object-oriented design: assigning responsibilities to classes. Specifically, given a set of features (methods and attributes) and their dependencies (data dependencies between methods and attributes and functional dependencies between methods) we are tasked to find:

1. A set of classes with unique names; and
2. An allocation of features to these classes that minimises dependencies between classes.

## 3 Overview of MDEOptimiser

MDEOptimiser[2] has been developed to allow expressing search and optimisation problems in an MDE context. It considers search spaces to be described by meta-models so that candidate solutions are represented by individual models [4]. Search-space exploration is performed by deriving new models from existing models; these transitions are encoded using Henshin transformations. Objective functions can be expressed as model queries or by giving arbitrary Java code (e.g., by calling out to an external simulation engine). MDEOptimiser provides an Xtext-based domain-specific language (DSL) for expressing an optimisation problem using these basic ingredients and a Java framework for running optimisation algorithms and generating a (near-)optimal result model (or population of models for multi-objective problems).

## 4 Solution Overview

We separate the problem into two sub-problems, which we are going to solve sequentially:

1. Find an optimal allocation of features to a suitable set of classes;
2. Ensure all classes have unique names.

[1] `https://github.com/martin-fleck/cra-ttc2016/`
[2] `https://github.com/szschaler/mde_optimiser`

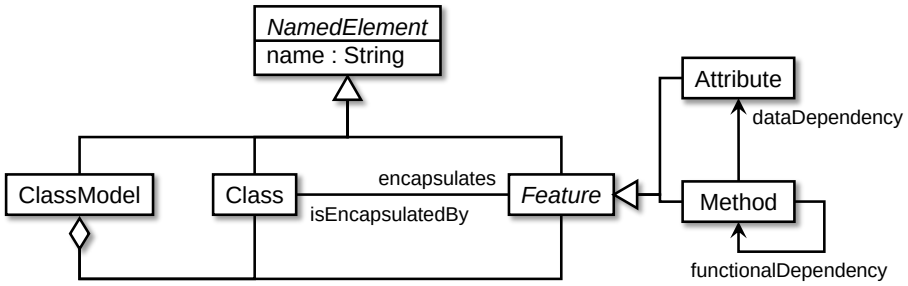


Figure 1: Metamodel describing the search space of the class–responsibility assignment problem

The first problem is a search problem, which we will solve with MDEOptimiser. The second problem is a simple transformation problem, which we solve using a simple iteration expressed in Xtend. In the following two sub-sections, we discuss each solution in turn. The complete solution is also available on SHARE [1]. The code of the solution is also available on GitHub.[3]

## 4.1 Solving the search problem

Three things are needed for any search problem:

1. A definition of the search space and a corresponding encoding for individual candidate solutions;
2. A means of exploring the search space by moving from existing solution candidates to new ones; and
3. A set of objective functions enabling the comparison of candidate solutions along a number of dimensions.

In this section, we describe each of these aspects for our solution using MDEOptimiser.

### 4.1.1 Search space definition

Figure 1 shows the metamodel included with the problem description of the TTC 2016 challenge case. Problem instances are specified as instances of this metamodel containing no `Class` instances. The goal is to (a) create a suitable number of `Class` instances, and (b) allocate `Feature`s to these classes (using the `encapsulates` reference) so as to optimise cohesion and coupling.

MDEOptimiser runs optimisation directly on models. That is, individual candidate solutions will be encoded as instances of the metamodel in Fig. 1

[3] `https://github.com/szschaler/mdeoptimise_ttc16`

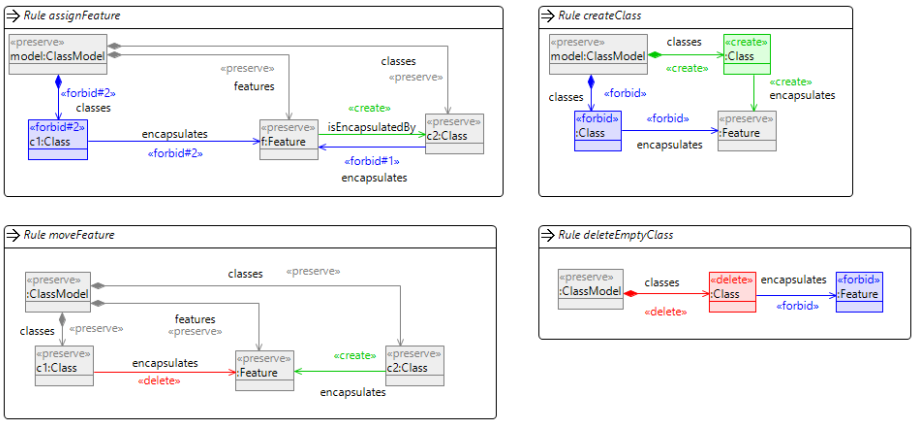


Figure 2: Model evolvers used for the challenge case

without using any other encoding. Thus, the metamodel provides us with a complete definition of the search space.

### 4.1.2 Model evolvers

MDEOptimiser uses Henshin transformation rules to specify how to derive new candidate solutions from given ones. At this point, the tool only supports "mutation"-type derivation of new candidate solutions; that is derivations that take a single model and produce a single new model. We call these mutators "model evolvers". Figure 2 shows the model evolvers used for the challenge case. When the search is executed, the engine will randomly pick an applicable evolver every time a new candidate solutions needs to be derived.

These rules are similar, but not identical to the rules given with the challenge case. In particular, we made the following changes:

1. *No names.* The rules do not match against or modify the names of any model elements. Ensuring uniqueness of names will be performed as a separate processing step, explained in Sect. 4.2.

2. *Additional rules.* The challenge case only included two rules (creating a class and assigning a feature). This was sufficient for the MoMOT-based [2] implementation, which uses sequences of rule applications to encode candidate solutions. Therefore, they have access to the 'transformation history' of any candidate model and can modify past transformation steps to find optimal solutions. In contrast, MDEOptimiser only keeps the model resulting from the transformation application. It cannot exchange a past transformation step, so needs additional evolvers to ensure it can fully explore the search space. As a result, we needed to add a rule for moving

features from one class to another. We also included a rule for deleting empty classes to enable the search to produce more compact models.

3. *Additional negative application conditions.* We found that we needed to introduce additional negative application conditions. In particular, it was necessary that rule `assignFeature` would only match against unassigned features. This is not automatically implied by marking the `isEncapsulatedBy` edge as `«create»`. Instead, we need to specify an explicit `«forbid»` edge to create a negative application condition. A peculiarity of the Henshin diagram editor means that we cannot create such an edge directly for `isEncapsulatedBy`, but instead had to define it for the inverse edge `encapsulates`.

4. *No simple class creation.* The original `createClass` rule, which created an empty class turned out to be inefficient. By changing it to a rule which creates a class and immediately assigns a previously unassigned feature to it, the search became substantially more efficient.

#### 4.1.3 Objective functions

As candidate solutions are encoded as models, objective functions can simply be encoded as model queries for the challenge case. We used the following objective functions:

- Minimise number of unencapsulated features;
- Minimise number of empty classes; and
- Maximise CRA (a combination of cohesion and coupling metrics as defined in the challenge case).

In principle, all of these metrics could be expressed in OCL. Our present prototype does not yet support expression of objective functions directly in OCL so that we had to write them in Xtend code, resulting in somewhat more cumbersome expressions than necessary. Integrating proper OCL support is an important piece of future work for MDEOptimiser.

#### 4.1.4 Putting it all together

Figure 3 shows how the CRA problem is specified in MDEOptimiser. In Line 3, we define the search space by indicating the relevant meta-model. Lines 5 to 7 define the fitness functions to be used and Lines 9 to 12 indicate how candidate solutions can be evolved into new candidate solutions. Figure 4 shows Xtend code to execute the search based on this model. We first instantiate a model provider, which in our case simply provides the one model that the current case is working on. Next, we instantiate the MDEOptimiser `OptimisationInterpreter`, providing the model from Fig. 3, a specific search algorithm (here a simple variant of non-dominated sorting based search), and

```
1  basepath <src/uk/ac/kcl/mdeoptimise/ttc16/models>
2
3  metamodel <architectureCRA.ecore>
4
5  fitness "uk.ac.kcl.mdeoptimise.ttc16.implementation.MinimiseClasslessFeatures"
6  fitness "uk.ac.kcl.mdeoptimise.ttc16.implementation.MinimiseEmptyClasses"
7  fitness "uk.ac.kcl.mdeoptimise.ttc16.implementation.MaximiseCRA"
8
9  evolve using <craEvolvers.henshin> unit "createClass"
10 evolve using <craEvolvers.henshin> unit "assignFeature"
11 evolve using <craEvolvers.henshin> unit "moveFeature"
12 evolve using <craEvolvers.henshin> unit "deleteEmptyClass"
```

Figure 3: Specification of the CRA optimisation problem in MDEOptimiser

```
90  val model = modelLoader.loadModel("src/uk/ac/kcl/mdeoptimise/ttc16/opt_specs/" + optSpecName +
91      ".mopt") as Optimisation
92
93  val modelProvider = injector.getInstance(CRAModelProvider)
94  modelProvider.setInputModelName(inputModelName)
98
99  val interpreter = new OptimisationInterpreter(model, new SimpleMO(1000, 50), modelProvider)
100 val optimiserOutcome = interpreter.execute()
101
102 // Ensure all classes have unique names
103 optimiserOutcome.map[cm|cm.getFeature("classes") as EList<EObject>].flatten.forEach [ cl, i |
104     cl.setFeature("name", "NewClass" + i)
105 ]
```

Figure 4: Invocation of MDEOptimiser for the CRA problem

the model provider. Invoking `execute` on this interpreter finally runs the search and returns the results.

As we are using a population-based algorithm, as is typical for multi-objective problems like CRA, we receive a population of models from which we will need to pick one. Some of the models in the population may not be valid as they may contain unassigned features (we treat feature assignment only as an objective function during the optimisation). From the remaining solutions, we pick the one that has the best CRA value.

### 4.2 Post-processing

Finally, we need to ensure all classes in the final model have unique names. This can be easily achieved by a simple post-processing step as shown in Fig. 4 (Lines 102*ff.*): we iterate over the list of classes in the models generated and set their names to a unique string by appending a running counter.

## 5 Evaluation

In this section, we evaluate our solution against the criteria defined in the challenge case, namely completeness & correctness, optimality, complexity, flexibility, and performance. We divide these into product criteria and process criteria, which we discuss in the following two sub-sections.

### 5.1 Product criteria

In this section we evaluate our transformation against the problem models given in the challenge case and report the results of the metrics for completeness & correctness, optimality, and performance. As search-based algorithms involve a certain amount of randomness, we have run the transformation 10 times on each input model and report the average time taken as well as the CRA for the best model found across these ten attempts.

It is worth noting that there can be quite substantial variation in the quality of the models produced. It seems that our optimiser currently gets easily stuck in local optima. Investigating the precise reasons for this remains as future work, but three candidate issues present themselves for initial investigation:

1. *Inefficient search algorithm.* We currently use a very simple search algorithm based on ideas from non-dominated sorting. It is very possible that this algorithm is inefficient. We are planning to replace this hand-coded implementation with standard implementations as, for example, available in the MOEA framework[4].

2. *No support for breeding:* Breeding—that is, creating a new candidate solution by intermixing aspects of two good parent solutions—is not yet supported in MDEOptimiser. This makes it more difficult for the search algorithm to escape from local maxima and means that there may be large parts of the search space that are never reached.

3. *Lack of randomness in Henshin's rule application algorithm:* Our experiments clearly show that Henshin deterministically picks a match to apply when there is more than one possibility. As a result, the search algorithm is more likely to go over the same ground again and again, making it less efficient.

Table 1 shows an overview of the results obtained for each of the input models provided. We ran two configurations for each model: First, we ran a search with 100 generations and a population size of 100 models (Configuration I). Second, we ran a search with 1,000 generations and a population size of 50 models (Configuration II). Configuration I uses the same parameters as Fleck's original solution[5], so we show their CRA values for comparison.

Note that for the less complex models we could already obtain reasonable results with a smaller number of generations and a smaller population size. For models B and C, our CRAs in Configuration I are better than Fleck's results. For model A, we are very close to their CRA value. Larger populations or more generations did not improve these values. Indeed, it appears from the table that we get worse results for Configuration II for models B and C. We believe that this is a result of the search getting stuck in local optima, helped by the fact that a smaller population size means more potentially interesting search routes are

[4] `http://moeaframework.org/`

[5] `https://github.com/martin-fleck/cra-ttc2016/blob/master/MOMoT_solution/TTC2016_CRA_MOMoT.pdf`

Table 1: Product-criteria measurements

| Model | Configuration | Avg. time | Best CRA | Fleck |
|---|---|---|---|---|
| A | I | 60s 751.2ms | 1.6667 | 1.75 |
| | II | 5m 03s 437.5ms | 1.6667 | |
| B | I | 4m 14s 603.7ms | 1.2667 | -.2333 |
| | II | 10m 30s 907.1ms | 1.0944 | |
| C | I | 6m 46s 346.5ms | 0.3363 | -6.4714 |
| | II | 29m 23s 386.2ms | -1.1137 | |
| D | I | 8m 01s 991.6ms | -44.3097 | -23.6338 |
| | II (over 6 runs) | 1h 47m 17s 675.0ms | -3.684490 | |
| E | I | 9m 50s 966.5ms | N/A | -66.6555 |
| | II | N/A | N/A | |

weeded out earlier. However, for model D, Configuration II produces a result with a much improved CRA, but at a substantial time cost. For model E, we were unable to complete the full experimental run by the submission deadline, as each individual run can take over four hours to complete. In an earlier run (with a slightly older version of the code base) the tool found a model with a CRA of $-0.4556$ after 4h 19m 10s 509ms (1,000 generations of 50 models). We have yet to recreate this scenario with the current code base.

## 5.2 Process criteria

The **complexity** of our solution is comparatively low as MDEOptimiser is a tool dedicated to the expression and execution of search problems. Consequently, the CRA challenge case is a very natural problem for our tool to tackle. In particular, all that is needed is:

1. A model provider, which loads the initial models and makes them available to the optimiser;
2. A set of Henshin rules specifying how candidate solutions can be derived from existing candidate solutions;
3. A set of objective functions written by implementing a specific Java interface; and
4. A specification of the optimisation problem as a whole, expressed in our own dedicated domain-specific language.

All of these parts are quite straightforward to write, as is the code required to combine them and trigger execution of the actual search. Perhaps the most complex bit to get right is the set of Henshin rules; we found that we needed to make a substantial number of changes over the originally provided set of rules to ensure a reasonably efficient exploration of the search space. Good debugging facilities would have been helpful in this process, but remain for future work.

Given that our tool is a very early prototype, there are a number of *accidental* complexities that make expressing search problems a little more difficult than strictly necessary. In particular, we do not currently support objective functions to be expressed directly as OCL model queries, requiring them to be expressed in Java instead. However, by using Xtend and a number of simple helper functions, we have made the expression of these queries sufficiently easy to be workable for this case study.

Being completely declarative, our solution is also quite **flexible:** Adding a new objective function simply requires adding a class implementing the corresponding interface and referencing it from the optimisation specification. Similarly, additional rules for evolving candidate solutions can be added quickly and easily.

# 6 Lessons learned

We have learned a number of lessons from applying MDEOptimiser to the TTC'16 challenge case. These lessons will influence our future work on MDEOptimiser:

- *Avoiding local optima.* Our current implementation seems to get easily stuck in local optima. We have already discussed in Sect. 5.1 a number of things we will investigate to avoid this.

- *Additional features required.* Being a very new tool, MDEOptimiser is missing a number of important features. For example, we are currently not supporting Henshin rule parameters, which would have given us an opportunity to ensure name uniqueness in one go. Similarly, we do not yet support constraints for the specification of valid solutions. Constraints may have made the search more efficient, in particular for the more complex input models where a large proportion of the final population consists of invalid models with unassigned features.

- *Systematic development of optimisations.* Not having support for rule parameters in the tool forced us to separate the overall transformation into two phases. It could be argued that this is actually a more sensible design. This raises the issue of identifying techniques for systematic development of optimisation-based model transformations.

- *Debugging and testing support.* Debugging and testing optimisation-based transformations is particularly difficult because they create a very large number of intermediate models and because of their stochastic nature. Current techniques for debugging and testing transformations provide only insufficient support for this type of transformation.

- *Differences between evolution rules required for different search techniques.* In solving the challenge case, we had to develop Henshin rules that differ significantly from the ones presented in the challenge case. Some of these

differences seem to be because our search algorithm works on models directly rather than on transformation chains. It will be important to better understand how differences in the search algorithm affect the shape of the transformation rules required.

# 7 Conclusion

In this paper, we have reported on our solution to the 2016 TTC challenge case on class–responsibility assignment using our tool MDEOptimiser. While it was clear from the start that our very new tool would find the problem challenging, we have been encouraged by the solution we have been able to present as well as by the relatively short amount of time it has taken us to put this together. It is clear that much remains to be learned, understood, and improved. However, these are promising indicators of the potential of running search and optimisation algorithms directly on models. We have been able to identify a number of lessons learned as well as challenges to inform our future work on MDEOptimiser.

# References


[1] Alexandru Burdusel and Steffen Zschaler. Online mdeoptimiser demo. `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=ArchLinux64_MDE-Optimiser.vdi`, 2016.

[2] Martin Fleck, Javier Troya, and Manuel Wimmer. Marrying search-based optimization and model transformation technology. In *Proc. 1st North American Search Based Software Engineering Symposium (NasBASE'15)*, 2015. Preprint available at `http://martin-fleck.github.io/momot/downloads/NasBASE_MOMoT.pdf`.

[3] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[4] Steffen Zschaler and Lawrence Mandow. Towards model-based optimisation: Using domain knowledge explicitly, 2016. Under review.