Solving the Class Responsibility Assignment Case with Henshin and a Genetic Algorithm

Kristopher Born, Stefan Schulz, Daniel Strüber, Stefan John

Philipps-Universität Marburg, Germany,
{born,schulzs,strueber,johns}@informatik.uni-marburg.de

Abstract. This paper presents a solution to the TTC2016 challenge "The Class Responsibility Assignment Case". Our solution uses the Henshin model transformation language to specify genetic operators in a standard genetic algorithm framework. Due to its formal foundation based on algebraic graph transformation, Henshin is well-suited to specify fundamental change patterns for genetic operators in a declarative manner. Adopting a simple, widely used genetic algorithm, we focus on effective implementation strategies for the genetic operators as well as additional operations. We analyzed the impact on our implemented strategies on the given evaluation criteria. Without giving a definitive recommendation of how to configure our tool, we found a drastic impact of some configuration options on the runtime and quality of its results.

1 Introduction

Class Responsibility Assignment is one of the cases in the 2016 edition of the Transformation Tool Contest [1]. The general goal is to produce a high-quality object-oriented design, an aim that plays an important role in refactoring and programming language migration scenarios. The specific task is to partition a given set of features with dependencies between them into a set of non-empty classes. In the formulation provided by the case authors, the starting point is a *responsibilities dependency graph* (RDG), a model that specifies a set of methods and attributes; methods can reference other methods as well as attributes. The task is to specify a transformation from RDGs to simple class models so that the output class models exhibit desirable coherence and coupling properties. These properties can be measured using the CRA index, a metric that combines coherence and coupling into a single number, thus allowing to evaluate the quality of created class diagrams in a convenient manner.

In this paper, we present our solution for the Class Responsibility Assignment Case based on the Henshin model transformation language [2] and a standard framework for genetic algorithms [3]. The main idea is to use graph-based model transformation rules to specify the genetic operators included in the framework, *mutation* and *cross-over*, and further operations. The resulting specification is largely declarative and targets a high abstraction level. In particular, we show how Henshin's advanced features, such as rule amalgamation and application conditions, enable a compact and precise specification.

This specification aligns well with genetic algorithms, which provide a robust and well-proven foundational search-based framework. Genetic algorithms have been successfully applied to global optimization problems like scheduling [4] and engineering tasks [5]. In particular, their modularity, configurability, and ultimately their flexibility in encoding problem domains make them appealing for software engineering problems, such as the one considered in this paper. We specified all optimization steps using Henshin rules.

While Henshin has been used in the context of search-based software engineering before [6], the distinctive feature of our solution is a set of specialized strategies addressing the Class Responsibility Assignment Case. We provide custom strategies for the implementation of the genetic operators, the creation of the initial population, and domain-specific post-processing operations. As we show in our preliminary evaluation based on the provided input models, these strategies have a substantial effect on the runtime of the algorithm and the quality of the produced result. We provide our solution using the SHARE platform [7]; the source code is available at BitBucket [8].

The rest of this paper is structured as follows. In Sec. 2, we introduce the necessary background. In Sec. 3, we present our solution. In Sec. 4, we show and discuss our evaluation results. In Sec. 5, we conclude.

2 Background

2.1 Henshin

Henshin [9, 2] is a model transformation language for the Eclipse Modeling Framework. It provides a visual editor for the specification of model transformations, an API for the execution of these transformations, and tool support for analyses such as state space exploration and critical pair analysis.

The Henshin language is based on the paradigm of algebraic graph transformations. In this paradigm, the basic building blocks for the definition of model transformations are *rules*. A rule comprises a *left-hand side* (LHS), specifying a graph pattern to be matched in the input model, and a *right-hand side* (RHS), specifying an intended change in case that the LHS can be matched. The LHS can be extended using *positive* and *negative application conditions* (PACs and NACs) to require the presence or absence of additional patterns in the input model. A rule can contain a set of *multi-rules*. Each multi-rule specifies a change to be executed as often as possible, effectively providing a *for-each* operator.

In its visual representation, the whole rule is shown as a single graph with annotated elements. Elements tagged with ⟨⟨delete⟩⟩ are only present in the LHS; that is, they are removed by the transformation. Elements tagged with ⟨⟨create⟩⟩ represent elements only part of the RHS, i.e., they are newly created. The annotation ⟨⟨preserve⟩⟩ denotes elements part of the LHS and RHS, i.e., they are left as is when applying the rule. Elements tagged with ⟨⟨forbid⟩⟩ specify NACs, preventing the rule from being executed when found. For multi-rule elements, the tag is augmented with an asterisk, for instance to ⟨⟨preserve*⟩⟩. We illustrate these concepts in Sect. 3; more examples are provided online [10, 11].

2.2 Genetic Algorithms

Genetic algorithms are a search-based optimization heuristic mimicking natural evolution processes. In contrast to traditional optimization strategies (e.g. hill climbing) genetic algorithms apply optimization to a whole *population* of possible solutions in parallel. This exploration of several locations of the search space combined with means for preserving the populations diversity facilitates a high level of robustness against premature stagnation at local optima.

Following Darwin's theory of natural selection, the optimization takes place by a stepwise creation of new generations. A *fitness function* is used to quantify the quality of each individual solution of the population. Adhering to the concept of survival of the fittest, good solutions are mated to exchange their valuable properties while the black sheep of the herd are left behind. In addition to this reproduction step, similar to mother nature, random mutations may occur from time to time. The created offspring form a new, hopefully fitter generation of solutions which will be the basis for the next optimization run.

While genetic algorithms in general are modular, extendable and highly configurable, the above concepts can be implemented by a simple genetic algorithm as described by Goldberg [3]. Apart from the mandatory fitness function such a basic implementation comprises three components:

- a *mutation operator* ensuring the diversity of the population by randomly altering individuals,
- a *crossover operator* simulating the natural process of mating individuals and exchanging their genetic information,
- and a *selection mechanism* responsible for enforcing survival of the fittest.

3 Solution

We implemented our solution on top of a generic framework for genetic algorithms available at GitHub [12], providing custom implementations for the initialization step, the mutation and crossover operators, and the fitness function used during the crossover and selection phases.

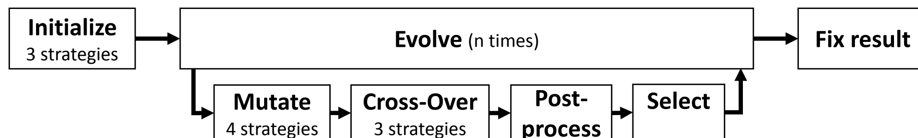

Fig. 1. Overview

Fig. 1 shows a high-level overview. During initialization, the starting population is generated. When the initialization is finished, evolution is started, using the maximum number of evolution steps n as an input parameter. To produce new individuals, each evolution step proceeds in four stages: First, the

individuals are mutated. By that, the number in the intermediate population of result models doubles. Second, during cross-over, the strongest individuals are combined with randomly chosen ones from the whole population. Third, the resulting individuals are post-processed. Finally, the best ten percent of all individuals and additional randomly chosen ones are selected as result of the evolution step. The genetic algorithm terminates after the n-th evolution step, followed by a step called *fix results*. As fitness function, we applied the CRA index, using the implementation provided along with the case description [1].

We now describe our solution in detail. In Sec. 3.1, we describe our three strategies to generate a starting population. In Sec. 3.2, we describe our four mutation strategies, involving the rearrangement of feature encapsulations and class creations and deletions. In Sec. 3.3, we describe our three crossover strategies, mainly differing by their mixture of randomness and preservation of existing properties. In Secs. 3.4 and 3.5, we describe *post-processing* and *fix result*.

3.1 Initialization

The goal during initialization is to obtain a set of class models that can be manipulated during the evolution phase. Since the input models initially arrive in the form of a RDG, basically a set of features, the goal is to ensure that each feature is encapsulated by a class. Please note that we do not consider any additional validity requirements until after the evolution phase (see Sect. 3.5).

Strategies We provide three strategies to establish that each feature is assigned to a class. The first is to create one dedicated class for each feature. The second is to create one class and assign *all* features to it, rendering it a "God class". The third is a combination of the first two strategies. We first describe our implementation of the first strategies, then how we ensure that *multiple* different input models can be produced, which eventually leads us to our third strategy.

The rules for strategies 1 and 2, shown in Fig. 2, harness Henshin's multi-rule concept (see Sect. 2.1). Rule *createClassPerFeature* creates a new class for each feature in the class model and encapsulates the feature in that class. In the resulting model, there are as many features as there are classes. In other words, the resulting model contains the maximum number of classes, considering only models with non-empty classes. Rule *allFeaturesInOneClass* creates a single "God class" in the class model. All features get encapsulated in that class.

Usually, a start population comprises multiple models. The number of individuals can be configured by setting a *population size*. The population size remains constant throughout the complete algorithm. This value is an influential factor for the runtime and the quality of the results. In both strategies, we produce variants of the initial input model by performing one random mutation step (see Sec. 3.2), a typical method to produce an initial population. To explore the solution space more broadly, we provide a third strategy, *mixed*, that produces m models by applying the first strategy to produce the first $\lceil \frac{m}{2} \rceil$ models and the second strategy to produce the remaining ones. For instance, to establish a population size of 10, it produces 5 variants of the God-class model and 5 variants of the class-per-feature class model.

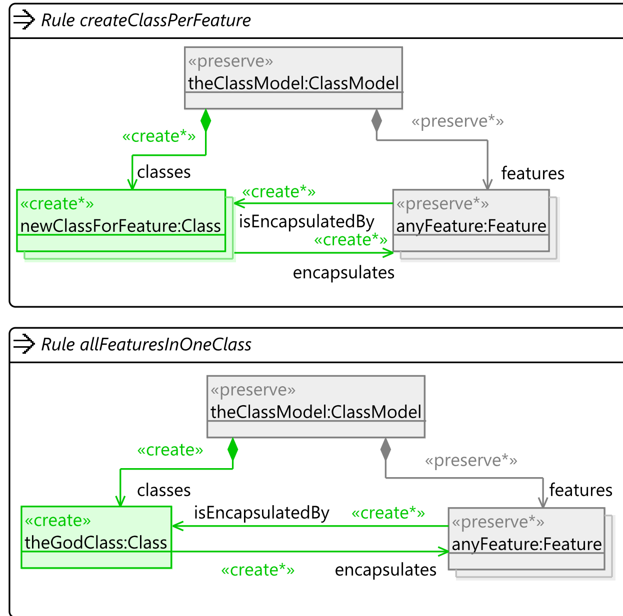

Fig. 2. Rules to create the initial population

3.2 Mutation

A mutation is one of the two genetic operators to produce new individuals. The mutation of a single individual may range from tiny to tremendous changes with a significant effect on the fitness score. While small changes may advance the approximation of a local maximum, major changes can provide access to new regions in the solution space that can help discover another maximum.

Strategies We specified four mutation strategies using the rules depicted in Fig. 3. The first three correspond to one of the rules each; the fourth strategy is produced from a combination of multiple rules. Our strategies are not mutually exclusive. In our evaluation, we experimented with all 16 possible combinations.

The rule *joinSelectedClasses* joins two classes. To this end, it moves all features from the to-be-deleted class to the remaining class. In addition, the deletion of the containment reference removes to-be-deleted class from the class model.

The rule *moveSelectedFeature* moves a single feature between two classes by deleting and creating its encapsulation references. A single application of this rule yields a minimal change, which would require many mutations to explore a wider area in the state space, especially for big input models. To accelerate this process, the rule is applied a random number of times during a single mutation.

The rule *moveAttributeToReferencingMethod* moves an attribute referenced by a method to the method's class, unless the attribute is referenced by another method in its own container class. Note that, in contrast to the first two mutations, this mutation is not a "blind" one, but designed to intuitively improve fitness by advancing cohesion and reducing coupling.

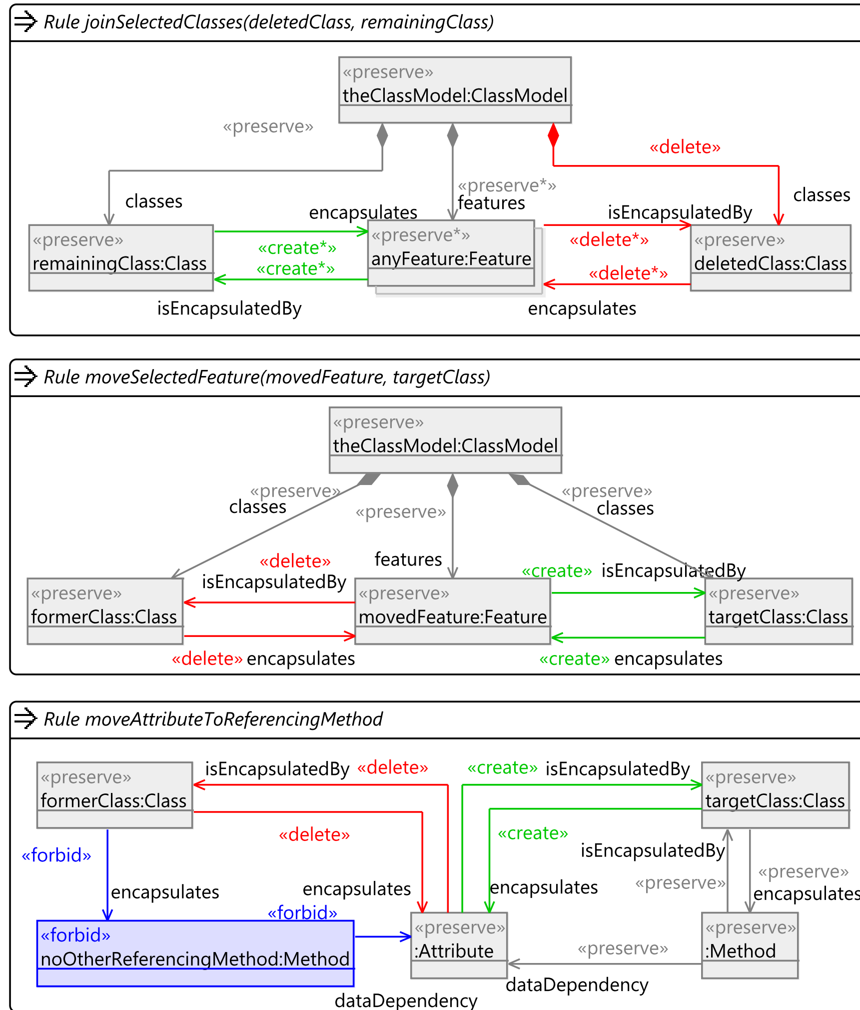

Fig. 3. Rules for the mutation operator.

The fourth mutation *randomSplitClass* splits a single class in several new ones and randomly distributes the features of the former class among them, so that each new class obtains at least one feature. The mutation consists of two elementary rules, *createClass* (depicted in Fig. 4) and *moveSelectedFeature*. We orchestrate these rules programmatically.

3.3 Crossover

Crossover is the second genetic operator. The core principle is to take two parent solutions and create a child from their combined genetic material. In our case, we

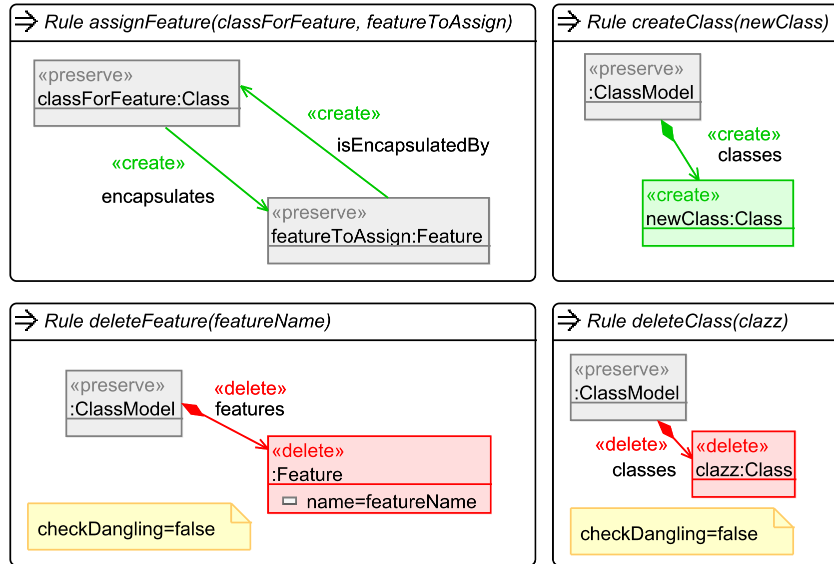

Fig. 4. Utility rules for crossover (and mutation) operators.

can cross two parent models by alternately selecting a class in one of them and copying that class to the child model. The feature assignment of the class is reproduced in the child model. To keep all three models in sync, features are deleted from the parent models immediately when they are assigned to a class in the child model. This process is repeated until no features remain in the parent models and each feature in the child model is assigned to a class. The original parent models are then restored to their original state to enable further mating.

Strategies. In our case, since the genetic material of the parent models is directly represented, it is tempting to "breed" children with desirable features. Our crossover strategies differ in the degree in which they rely on this idea. Each strategy determines which classes are selected during mating. First, in *randomClassCrossover*, the class to be reproduced in the child is chosen randomly. Second, in *classWithBestCohesionCrossover*, the classes with the best cohesion value are selected, ignoring coupling. Third, *classWithBestCohesionAndCouplingCrossover* considers coupling as well. However, it is important to notice that cohesion and coupling values in the parent models are not directly transferable to the child model. The reason is that the parents are changed within the process; the resulting values in the newly created class model will differ.

We have implemented these strategies using the simple rules shown in Fig. 4, orchestrating them in our Java implementation. The rules *deleteFeature* and *deleteClass* are configured in a way that disables the default check for dangling edges. This setting defines whether a transformation is applied or not if the transformation would leave behind dangling edges in the context of element deletions. The shown rules delete features and classes even if they have incoming or outgoing edges, which are removed automatically by the Henshin interpreter.

3.4 Post-processing

In a dedicated step before the selection of the fittest individuals, we can harness domain knowledge to improve the candidate individuals. Specifically, the mutation rule *moveAttributeToReferencingMethod* shown in Fig. 3 improves the fitness rating in most cases, as we have observed in our experiments. We provide a configuration option to apply this mutation on each individual created during an evolution step. In the rare case that this optimization produces a less fit individual, the optimization is ignored during selection.

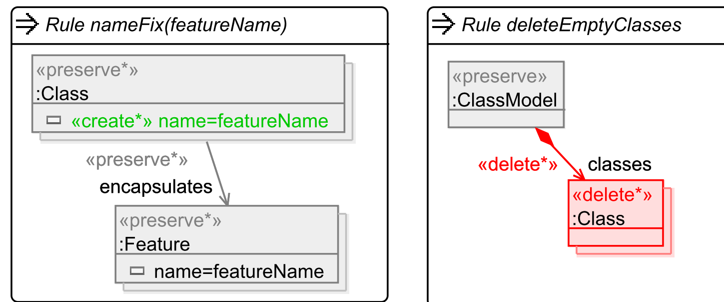

Fig. 5. Rules for *fix result*.

3.5 Fix result

The goal of *fix result* is to turn the result model into a valid class model, satisfying the constraint that each class must have a unique name. We noticed that the enforcement of this requirement is best postponed to a dedicated clean-up phase since it might otherwise interfere with the optimization.

To ensure that each class has a unique name, we apply the rules shown Fig. 5 on the result model. The *nameFix* rule comprises a multi-rule that iterates over all pairs of a class and an associated feature. For each of these pairs, the class name is set to the value of the feature name, by matching the feature's *name* attribute in the LHS, storing its value in a parameter called *featureName*, and propagating the value to the class name attribute in the RHS.

In general, it may occur at this point that the class model exhibits empty classes, that is, classes without an assigned feature. For instance, a class created during the *random split* mutation might not have obtained an associated feature. Rule *deleteEmptyClasses* specifies that all empty classes are deleted from the class model. It specifies this deletion using a multi-rule. The dangling condition (see Sec. 3.3) ensures that classes with an associated feature are not affected by this rule, since their deletion would leave behind dangling edges.

Since the remaining classes are generally non-empty, each class has finally obtained a name: that of one of its members, chosen nondeterministically, according to the principle of *"last write wins"*. Conversely, since each feature is assigned to exactly one class, the resulting class names are unique.

4 Preliminary Evaluation

In our evaluation, we investigated the impact of our different strategies, focusing on the quality of the produced results and the performance behavior during the creation of these results. We measured the quality in terms of the CPA index, as stipulated in the case description [1]. We determined performance behavior by measuring the runtime of the algorithm to produce the result models. To study the effects of our strategies in isolation, we varied the treatment among all possibilities within one category (initialization, mutation, cross-over, post-processing), using a fixed configuration for the remaining configuration parameters. In each case, we studied the effect on the provided example models 1–5. We ran all experiments on a Windows 7 system (3.4 GHz; 8 GB of RAM).

We used the following parametrization in our experiments: In all experiments, we used the same population size (5), number of runs (10), and post-processing configuration (activated). In the case of the initialization and cross-over strategies, we considered 20 evolution steps. In the case of mutation, we only considered 10 evolution steps, since the relevant configuration space was considerably larger. By visual inspection of barplots, we observed that these configuration were usually sufficient for the different runs in one experiment to converge. Runner classes with the full configurations are provided as part of our implementation [8], allowing the experiments to be reproduced with little effort.

4.1 Influence of Selected Initializations

To investigate the influence of the selected initializations we applied the three strategies described in subsection 3.1: *oneClassPerFeature*, *allFeaturesInOneClass* ("God class"), and *mixed*, a combination of the first two strategies.

Input models A and B: For input model A, in the *allFeaturesInOneClass* case, four evolution iterations are required to reach the optimal CRA of 3.0. In the *oneClassPerFeature* and *mixed* cases, the same value is always reached in the first evolution step. Similarly, for input model B, the optimal CRA value of 3.0 was reached in the first evolution step in the *oneClassPerFeature* and *mixed* cases. The *allFeaturesInOneClass* initialization strategy shows a flat development at a median CRA index of 1.9.

Input model C, D, and E: After 20 iterations, the CRA values for input model C were only negligibly different, amounting to a median of 1.0 for *allFeaturesInOneClass*, 1.1 for *oneClassPerFeature* and 0.9 for mixed. In all three strategies, an upward trend was emerging around the cut-off point. We observed a similar trend for input model D as well. In this example, it is important to notice that after the *oneClassPerFeature* initialization, a CRA of 0.95 in mean is reached, while the mean in both other cases amounts to 0.19. Finally, for input model E, with the *oneClassPerFeature* initialization, we observed the best mean value of 1.9, but the difference is small again, amounting to 1.7 in the *allFeaturesInOneClass* and 1.6 in the *mixed* case. Interestingly, at this point in the measurement, in all cases a similar number of classes is reached (around 5). A prolonged run could offer additional evidence in this case.

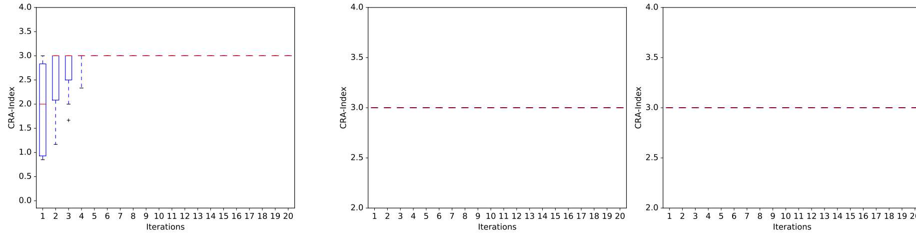

Fig. 6. CRA of input model A depending on the selected initialization *allFeaturesInOneClass*, *oneClassPerFeature* or *mixed* (from left to right).

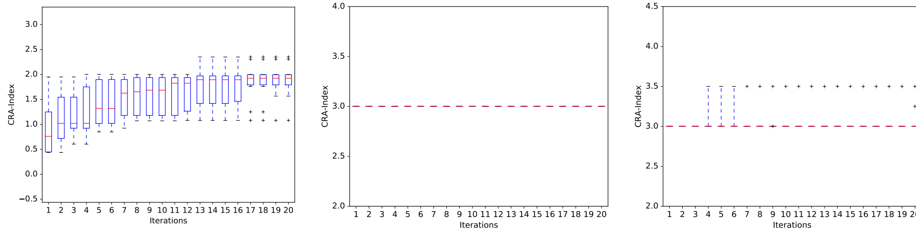

Fig. 7. CRA of input model B depending on the selected initialization *allFeaturesInOneClass*, *oneClassPerFeature* or *mixed* (from left to right).

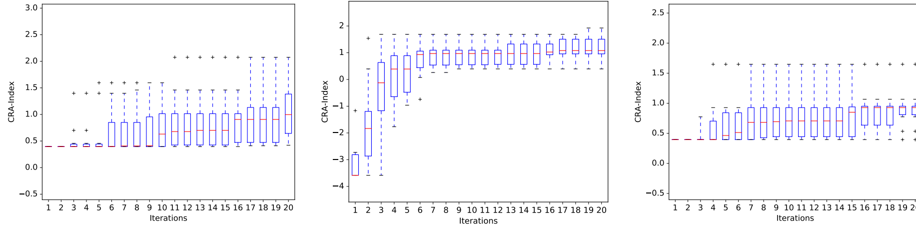

Fig. 8. CRA of input model C depending on the selected initialization *allFeaturesInOneClass*, *oneClassPerFeature* or *mixed* (from left to right).

In sum, *oneClassPerFeature* offered a moderate benefit for input models B and D, whereas all strategies were roughly on par in the other scenarios. While additional experiments with larger models and longer evolution runs are required for a more complete picture, we used *oneClassPerFeature* as initialization strategy in our further experiments.

4.2 Influence of Mutation Strategies

The effect of the mutation strategies is shown in Fig. 16. Since strategies in this category are orthogonal and can be combined, we experimented with each of the 16 possible combinations.

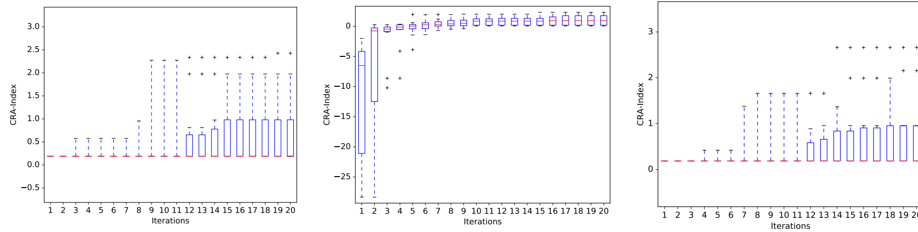

Fig. 9. CRA of input model D depending on the selected initialization *allFeaturesInOneClass*, *oneClassPerFeature* or *mixed* (from left to right).

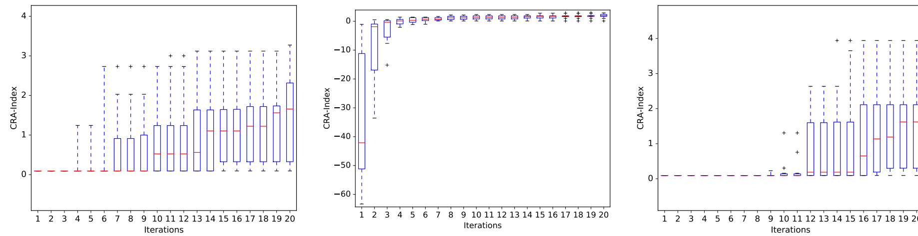

Fig. 10. CRA of input model E depending on the selected initialization *allFeaturesInOneClass*, *oneClassPerFeature* or *mixed* (from left to right).

Input models A and B. In the case of models A and B, the chosen mutation strategy did not affect the quality of the result; the CRA value was 3 in all executions. Even though the runtime for input model A took up to twice as long depending on the mutation strategy, the absolute runtime is still relatively low.

Input models C, D, and E. In the case of input models C, D, and E, a clear picture emerges. Based on their runtime behavior as well as the CRA index of the produced results, two clusters of mutation strategy combinations can be identified, a strong and a weak one. Remarkably, one of the strategies, *joinSelectedClasses* is contained in each of the strong combinations. A possible explanation of this observation is that none of the other strategies is suitable to reduce the number of classes to a significant extent, which becomes an important drawback in our chosen initialization strategy that creates a large set of classes. The CRA scores lay constantly in a range between 0 and 2.

4.3 Influence of Crossover Strategies

We studied result quality and runtime under varying crossover strategies, experimenting with the *random*, *coherence-* and *coherence/coupling-based* crossover strategies according to our description in Sec. 3.3. In addition, we studied the effect of deactivating the cross-over operator altogether.

We omit a visualization of our results here as we did not observe any differences that would justify a decisive judgement. In the case of models A and B,

Fig. 11. Input model A

Fig. 12. Input model B

Fig. 13. Input model C

Fig. 14. Input model D

Fig. 15. Input model E

Fig. 16. Effect of mutation strategies on CRA and runtime in the example models 1–5.

we measured CRA values of 3.0 for input model A and B right from the start. Therefore, the crossover strategy did not play any role at all. But even for input models C and D, the determined CRA values differed only marginally, usually amounting to values between 0.0 and 1.0. This also applies for the runs where the crossover strategy was deactivated.

In conclusion, it is indicated that our crossover strategies only make a minor contribution to the quality of the results. Even if its not possible to give a clear advice which crossover strategy to prefer, it is worth pointing out that the *classWithBestCohesionCrossover* strategy performed best for input model D while *classWithBestCohesionAndCouplingCrossover* gave the best result for input models C and E.

4.4 Discussion

While giving a definitive usage recommendation is outside the scope of this paper, our measurements give us a first preliminary reference for the configuration of our implementation. The initialization with *oneClassPerFeature* constantly achieved preferable CRA values. Regarding the selected mutation features, we strongly recommend to include *joinSelectedClasses*. Generally, activating the *moveAttributeToReferencingMethod* post-processing option has proven valuable. Finally, the selected crossover strategy turned out to be negligible in our experiments. Yet it remains unclear if this observation indicates a weakness of our crossover strategies or a strength of our initialization and mutation strategies. We are optimistic that comparing our implementation with other solutions to the Class Responsibility Assignment Case will lead to interesting insights in this respect.

5 Further Improvements

Despite a couple of first insights, we are only at the beginning of understanding the tuning of our technique. A longer series of experiments is required to provide more reliable evidence than given so far. Furthermore, while the transformation rules in our solution are simple, a formal proof that the produced models are always valid is left to future work. With regards to performance, the most evident improvement we see is based on the *embarrassingly parallel* nature of search-based techniques [13]. An implementation that distributes the individual rule applications across a multi-kernel architecture seems a suitable opportunity for a performance optimization.

References

1. J. T. Martin Fleck and M. Wimmer, "The class responsibility assignment case," in *Transformation Tool Contest*, 2016.

2. T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced concepts and tools for in-place EMF model transformations," in *Proc. Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'10)*, ser. LNCS, vol. 6394, 2010, pp. 121–135.
3. D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1989.
4. A. Kumar and A. K. Dhingra, "Optimization of scheduling problems: A genetic algorithm survey," *International Journal of Applied Science and Engineering Research*, vol. 1, no. 1, 2012.
5. P. M. S. Carvalho, L. A. F. M. Ferreira, and L. M. F. Barruncho, "On spanning-tree recombination in evolutionary large-scale network problems-application to electrical distribution planning," *Evolutionary Computation, IEEE Transactions on*, vol. 5, no. 6, pp. 623–630, 2001.
6. M. Fleck, J. Troya, and M. Wimmer, "Marrying search-based optimization and model transformation technology," *Technology. Proc. of the 1st North American Search Based Software Engineering Symposium*, 2015.
7. SHARE platform, strueber@informatik.uni-marburg.de, "Online demo: XP-TUe_TTC16_NMF_mrttc16.vdi," http://tinyurl.com/guteas4, 2016.
8. K. Born, S. Schulz, D. Strüber, and S. John, "Bitbucket: ttc16," https://bitbucket.org/ttc16/ttc16/overview, 2016.
9. Henshin, http://www.eclipse.org/modeling/emft/henshin.
10. Eclipse, https://wiki.eclipse.org/Henshin.
11. Henshin, https://www.eclipse.org/henshin/examples.php.
12. Y. Lahodiuk, https://github.com/lagodiuk/genetic-algorithm, generic implementation of Genetic algorithm in Java.
13. M. Harman, P. McMinn, J. T. De Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical software engineering and verification*. Springer, 2012, pp. 1–59.