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

Kristopher Born, Stefan Schulz, Daniel Strüber, Stefan John
Software Engineering Research Group, University Marburg
{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 transformations, 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 our implemented strategies on the given evaluation criteria, finding 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 [FTW16]. The general goal is to produce a high-quality object-oriented design, which 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 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 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 enabling the evaluation of the quality of created models in a convenient manner.

In this paper, we present our solution based on the Henshin model transformation language [ABJ$^+$10] and a standard framework for genetic algorithms [Gol89]. The main idea is to use graph-based model transformation rules to specify the genetic operators included in the framework, *mutation* and *crossover*, and further operations. The resulting specification is largely declarative. 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 such as scheduling [KD12] and engineering tasks [CFB01]. 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 [FTW15], the distinctive feature of our solution is a set of specialized strategies addressing the Class Responsibility Assignment

In: A. Garcia-Dominguez, F. Krikava and L. Rose (eds.): Proceedings of the 9th Transformation Tool Contest, Vienna, Austria, 08-07-2016, published at http://ceur-ws.org

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 (http://tinyurl.com/guteas4) and the source code at BitBucket (https://bitbucket.org/ttc16/ttc16/overview).

2 Solution

We implemented our solution on top of a generic framework for genetic algorithms available at GitHub (https://github.com/lagodiuk/genetic-algorithm), providing custom implementations for the initialization step, the mutation and crossover operators, and the fitness function used during the crossover and selection phases.

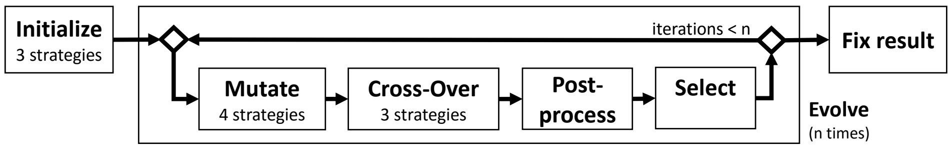

Figure 1: Overview

Figure 1 shows a high-level overview. During initialization, the starting population is generated. When the initialization is finished, the evolution consisting of n evolution steps is started. 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 crossover, 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, provided along with the case description [FTW16].

In Sec. 2.1, we describe our three strategies to generate a starting population. In Sec. 2.2, we describe our four mutation strategies, involving the rearrangement of feature encapsulations and class creations and deletions. In Sec. 2.3, we describe our three crossover strategies, mainly differing by their mixture of randomness and preservation of existing properties. In Secs. 2.4 and 2.5, we describe *post-processing* and *fix result.*

2.1 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 an 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. 2.5).

Strategies We provide three strategies to establish that each feature is assigned to a class. The used rules, shown in Figure 2, harness Henshin's multi-rule concept as indicated by the asterisk operator ($*$). The first strategy is to create one dedicated class for each feature. 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. The second strategy is to create one class and assign *all* features to it, rendering it a "God class". Rule *allFeaturesInOneClass* creates a single "God class" in the class model and encapsulates all features in that class. The third strategy, *mixed*, is a combination of the first two strategies to explore the solution space more broadly. It 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.

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 all strategies, we produce variants of the initial input model by performing one random mutation step (see Sec. 2.2), a typical method to produce an initial population.

2.2 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

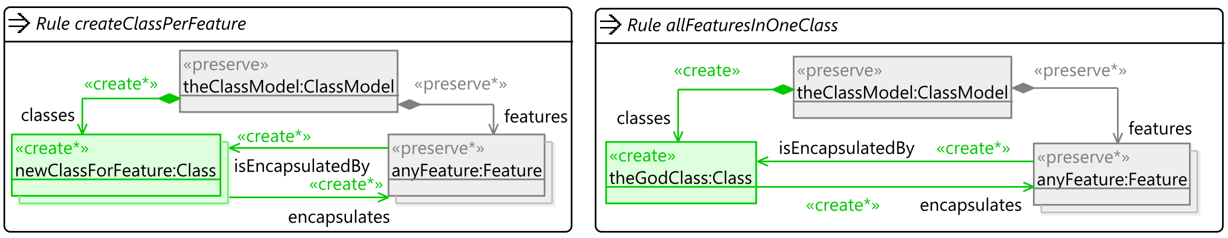

Figure 2: Rules to create the initial population

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 Figure 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.

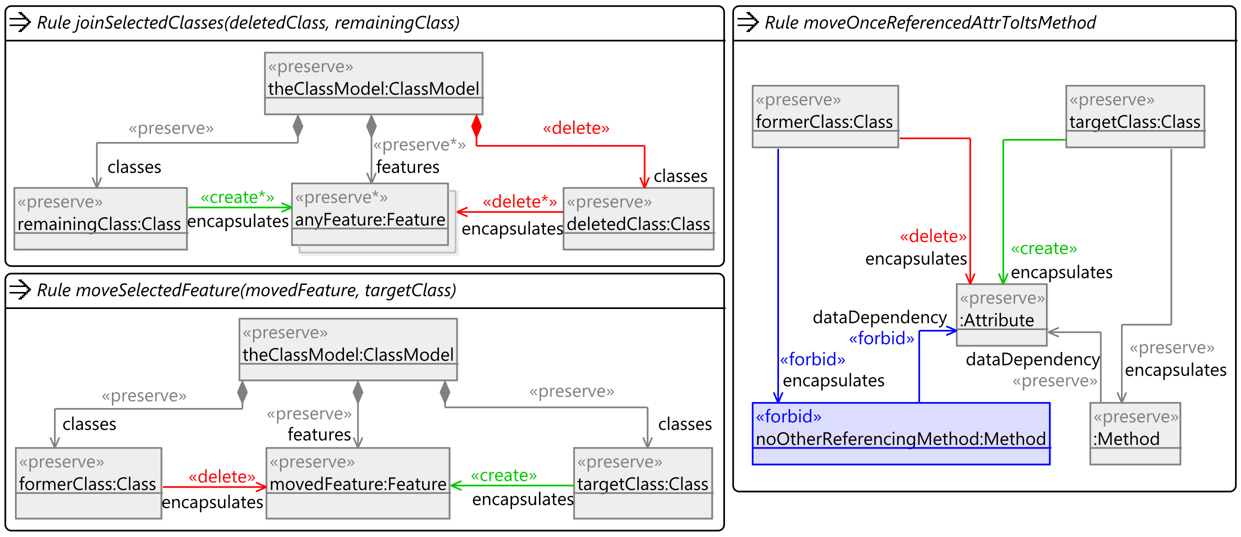

Figure 3: Rules for the mutation operator.

The rule *joinSelectedClasses* joins two classes. To this end, it moves all features from the *deletedClass* to the *remainingClass*. The deletion of the containment reference *classes* removes *deletedClass* 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 *moveOnceReferencedAttrToItsMethod* 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.

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 Figure 4) and *moveSelectedFeature*. Since Henshin's built-in control flow mechanism (units) lacks a concept to specify the application of a rule a random number of times, we orchestrated these rules programmatically.

2.3 Crossover

The key idea of crossover is to take two parent solutions and create a child from their combined genetic material. In our case, we can cross two parent models by alternately selecting a class in one of them and copying that class to the child model. Afterwards the feature assignment 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 in the child

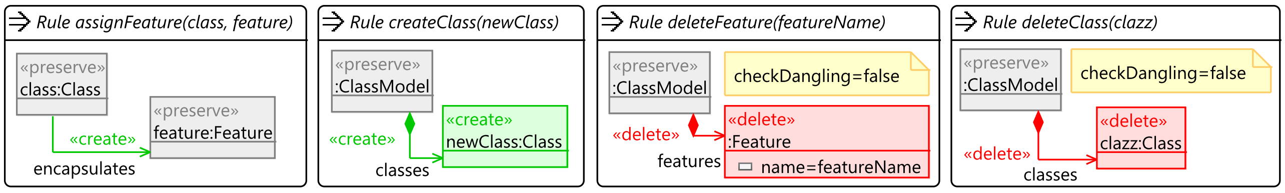

Figure 4: Utility rules for crossover (and mutation) operators.

model. This process is repeated until no features remain in the parent models and each feature in the child model is assigned. The original parent models are preserved by copying for 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 Figure 4, orchestrating them programmatically 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.

2.4 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 Figure 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.

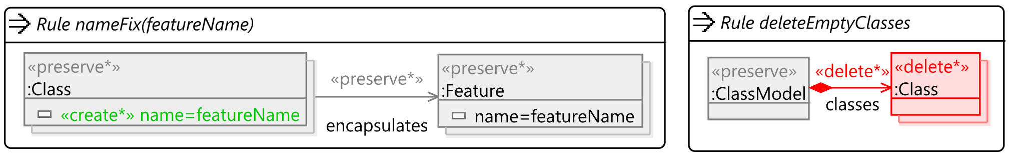

Figure 5: Rules for *fix result*.

2.5 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 Figure 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 implicit dangling condition (see Sec. 2.3) 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.

3 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 CRA index, as stipulated in the case description [FTW16]. 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, crossover, 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. The detailed results are in appendix A.

Based on the results of our evaluation we decided to configure the initialisation with *oneClassPerFeature*, to use all four mutations, activate post processing and chose the random crossover to achieve competitive results. In table 3 the results of two different runs are listed. The left part aims at rather low run-time based on 10 runs, 10 iteration per run and a population size of 5. The right part of the table demonstrates that our solution might provide even better results by the price of an increased run-time.

Table 1: Results of a fast and a slow run based on the recommended configuration

	10 runs, 20 iter/run, p_size: 5				40 runs, 40 iter/run, p_size: 7		
	CRA		run-time		CRA	run-time	
input	avg	best	avg	total	best	avg	total
A	3.0	**3.0**	228 ms	2.3s	**3.0**	0.67 s	26.9 s
B	3.05	**3.5**	519 ms	5.2s	**3.5**	1.58 s	63.4 s
C	1.17	**2.0**	981 ms	9.8 s	**2.2**	3.1 s	126.8 s
D	1.18	**2.6**	3.52 s	35.2 s	**3.3**	11.42 s	457.1 s
E	2.0	**3.9**	13.05 s	130.5 s	**5.7**	40.87 s	1635.2 s

4 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 our transformation steps are simple and the validity of the produced models has been ensured through application of the validation tool provided by the case authors, a formal proof that the produced models are always valid is left to future work. The most evident performance improvement we see is based on the *embarrassingly parallel* nature of search-based techniques [HMDSY12]. An implementation that distributes the individual rule applications across a multi-kernel architecture seems suitable for performance optimization.

Acknowledgements. This work was partially funded by the German Research Foundation, Priority Program SPP 1593 "Design for Future – Managed Software Evolution".

References

[ABJ+10] 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*, LNCS, 2010.

[CFB01] 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 Trans.*, 2001.

[FTW15] 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.

[FTW16] M. Fleck, J. Troya, and M. Wimmer. The class responsibility assignment case. In *Transformation Tool Contest*, 2016.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Professional, 1989.

[HMDSY12] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical software engineering and verification*, pages 1–59. Springer, 2012.

[KD12] A. Kumar and A. K. Dhingra. Optimization of scheduling problems: A genetic algorithm survey. *International Journal of Applied Science and Engineering Research*, 1(1), 2012.

A Detailed Results

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 crossover 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, allowing the experiments to be reproduced with little effort. We ran all experiments on a Windows 7 system (3.4 GHz; 8 GB of RAM).

A.1 Influence of Selected Initializations

To investigate the influence of the selected initializations we applied the three strategies described in subsection 2.1: *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.

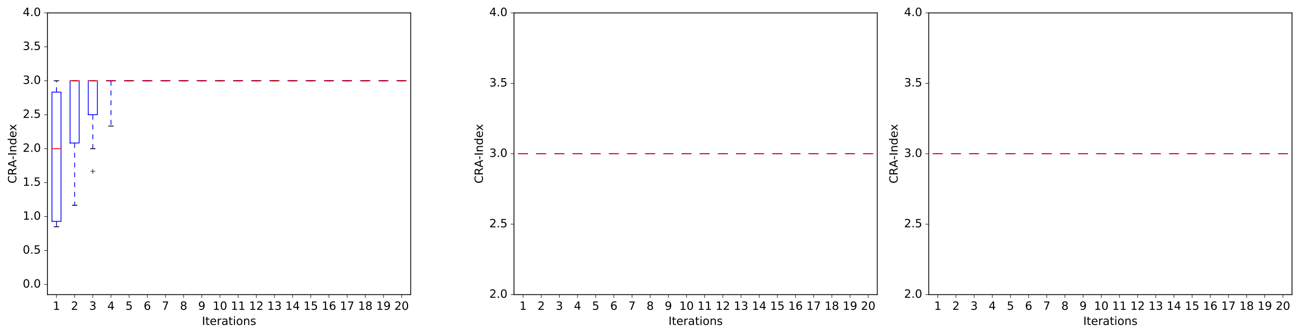

Figure 6: CRA of input model A depending on the selected initialization *allFeaturesInOneClass*, *oneClassPerFeature* or *mixed* (from left to right).

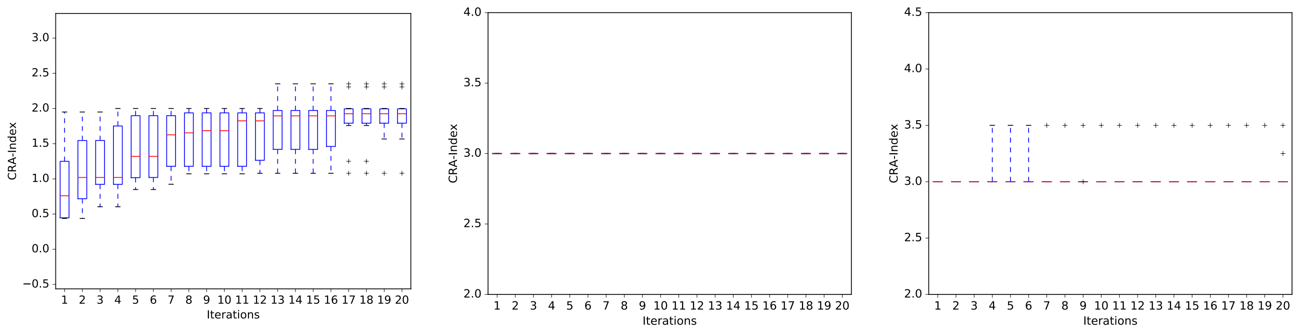

Figure 7: CRA of input model B depending on the selected initialization *allFeaturesInOneClass*, *oneClassPerFeature* or *mixed* (from left to right).

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

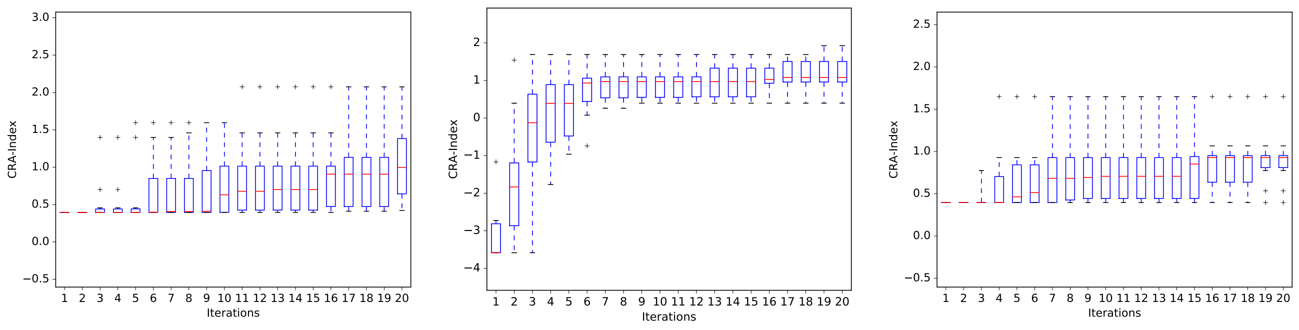

Figure 8: CRA of input model C depending on the selected initialization *allFeaturesInOneClass*, *oneClassPerFeature* or *mixed* (from left to right).

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.

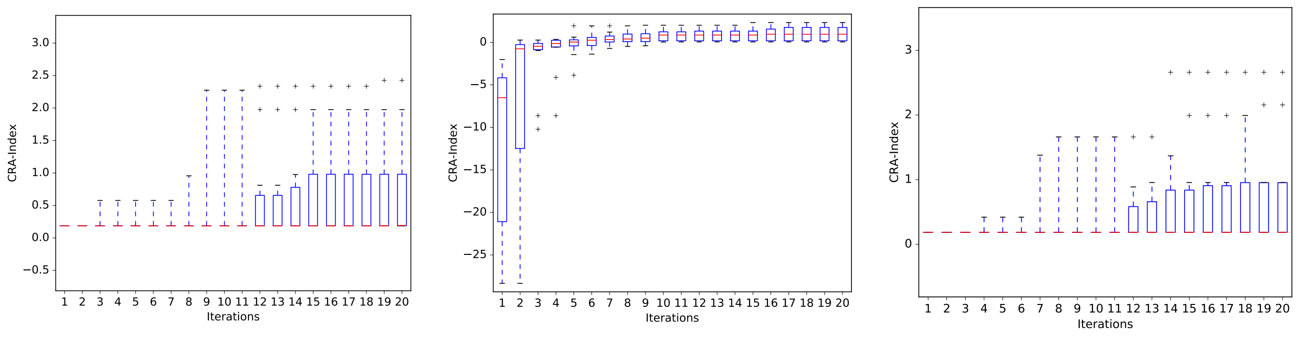

Figure 9: CRA of input model D depending on the selected initialization *allFeaturesInOneClass*, *oneClassPerFeature* or *mixed* (from left to right).

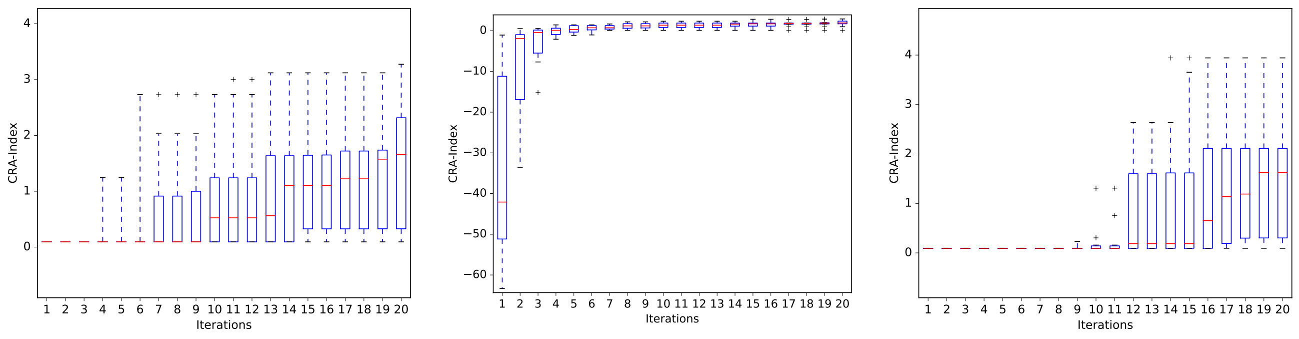

Figure 10: CRA of input model E depending on the selected initialization *allFeaturesInOneClass*, *oneClassPerFeature* or *mixed* (from left to right).

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.

A.2 Influence of Mutation Strategies

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

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.

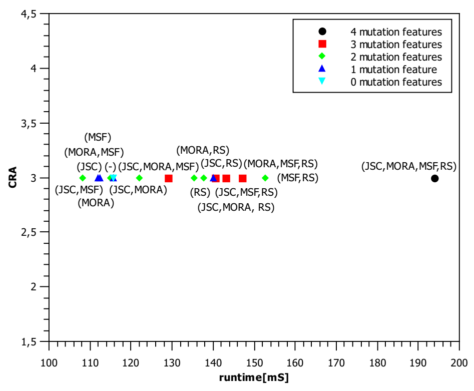

Figure 11: Input model A

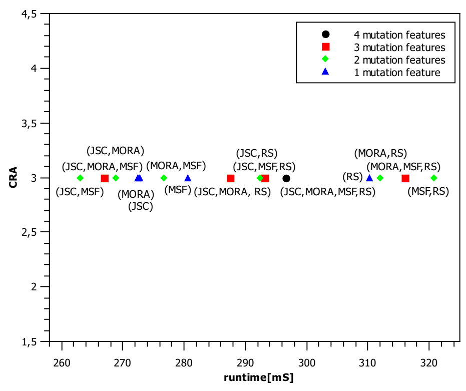

Figure 12: Input model B

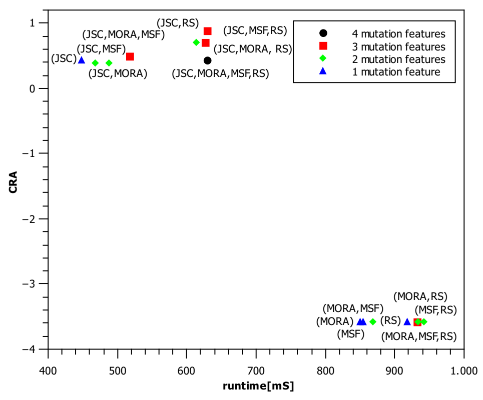

Figure 13: Input model C

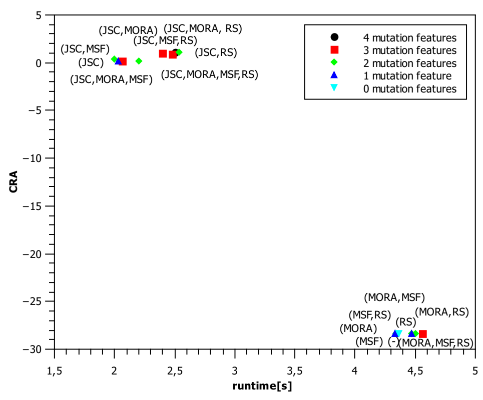

Figure 14: Input model D

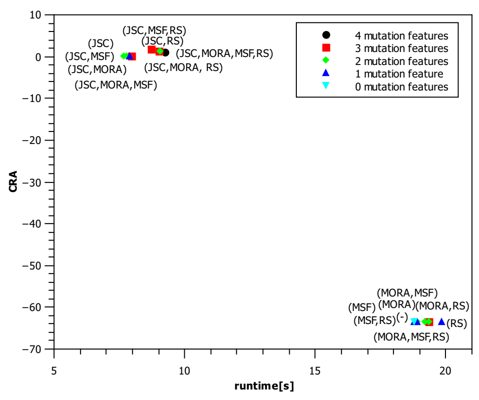

Figure 15: Input model E

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

A.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. 2.3. In addition, we studied the effect of deactivating the crossover 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, we measured CRA values of 3.0 for input model A and B right from the start. Therefore, the crossover strategy did not any play a role at all. But even for the 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.