Proceedings of the 9th International Workshop
Transformation Tool Contest

TTC 2016

Antonio Garcia-Dominguez
Filip Krikava
Louis Rose

Preface

The aim of the Transformation Tool Contest (TTC) series is to compare the expressiveness, the usability, and the performance of transformation tools along a number of selected case studies. A deeper understanding of the relative merits of different tool features will help to further improve transformation tools and to indicate open problems.

This contest was the ninth of its kind. For the fourth time, the contest was part of the Software Technologies: Applications and Foundations (STAF) federation of conferences. Teams from the major international players in transformation tool development have participated in an online setting as well as in a face-to-face workshop.

In order to facilitate the comparison of transformation tools, our programme committee selected the following challenging case via single blind reviews: Class Responsibility Assignment Case, for which eventually eight solutions were accepted.

These proceedings comprise descriptions of the three cases and descriptions of all of the solutions to these cases. In addition to the solution descriptions contained in these proceedings, the implementation of each solution (tool, project files, documentation) is made available for review and demonstration via the SHARE platform (http://share20.eu).

TTC 2016 involved open (i.e., non anonymous) peer reviews in a first round. The purpose of this round of reviewing was that the participants gained as much insight into the competitors solutions as possible and also to identify potential problems. At the workshop, the solutions were presented. The expert audience judged the solutions along a number of case-specific categories, and prizes were awarded to the highest scoring solutions in each category. A summary of these results for each case are included in these proceedings. Finally, the solutions appearing in these proceedings were selected by our programme committee via single blind reviews. The full results of the contest are published on our website[1].

Besides the presentations of the submitted solutions, the workshop also comprised a live contest. That contest involved creating a solution for dataflow-based model transformations. The live contest was announced to all STAF attendees and participants were given four days to design, implement and test their solutions. The contest organisers thank all authors for submitting cases and solutions, the contest participants, the STAF local organisation team, the STAF general chair Gerti Kappel, and the program committee for their support.

8th July, 2016
Vienna, Austria

Antonio Garcia-Dominguez
Filip Krikava
Louis Rose

[1] `http://www.transformation-tool-contest.eu/`

Organisation

Transformation Tool Contest has been organized by Antonio Garcia-Dominguez, Filip Kikava and Louis Rose.

Program Committee

Olivier Barais	University of Rennes 1, France
Philippe Collet	Universit Nice Sophia-Antipolis, France
Coen De Roover	Vrije Universiteit Brussel, Belgium
Antonio Garcia-Dominguez	Aston University, United Kingdom
Jeff Gray	University of Alabama, United States
Tassilo Horn	SHD, Germany
Akos Horvath	Budapest University of Technology and Economics
Christian Krause	SAP Innovation Center, Germany
Filip Kikava	Northeastern University
Sonja Schimmler	Bundeswehr University Munich, Germany
Arend Rensink	University of Twente, The Netherlands
Louis Rose	University of York, United Kingdom
Bernhard Schatz	Technische Universitat Munchen, Germany
Massimo Tisi	Ecole des Mines de Nantes, France
Tijs van der Storm	Centrum Wiskunde & Informatica, The Netherlands
Pieter Van Gorp	Eindhoven University of Technology, The Netherlands
Gergely Varro	Technische Universitat Darmstadt, Germany
Bernhard Westfechtel	University of Bayreuth, Germany

Additional Reviewers

Table of Contents

The Class Responsibility Assignment Case

Martin Fleck
Business Informatics Group
TU Wien, Austria
fleck@big.tuwien.ac.at

Javier Troya
ISA Research Group
Universidad de Sevilla, Spain
jtroya@us.es

Manuel Wimmer
Business Informatics Group
TU Wien, Austria
wimmer@big.tuwien.ac.at

Abstract

This paper describes a case study for the ninth Transformation Tool Contest (TTC'16)[1]. The case is aimed at the production of high-quality designs for object-oriented systems and presents the problem of finding a good class diagram for a given set of methods and attributes with functional and data relationships among them. In order to obtain such a class diagram, dedicated quality metrics that have been defined in the context of the class responsibility assignment problem need to be optimized. Therefore, the focus of this case study is not on the definition of the necessary set of rules, but rather on the orchestration of such rules in order to find the optimal class diagrams. The evaluation of the produced transformation is driven by the quality of the produced models, the complexity of the rule orchestration as well as by the flexibility of the solution and its performance.

1 Introduction

The quality of an object-oriented design has a direct impact on the quality of the code produced: the higher the quality of the models in the design, the higher the quality of the code. The Class Responsibility Assignment (CRA) problem [BBL10] deals with the creation of such high-quality object-oriented models. When solving the CRA, one needs to decide where responsibilities, under the form of class operations and attributes they manipulate, belong and how objects should interact. Therefore, CRA is a non-trivial problem, often requiring human judgement and decision-making [MJ14]. The necessity to assign responsibility to classes can arise in the context of two problems:

- *Finding a class diagram.* When migrating an application from a procedural language such as C to an object-oriented language, one needs to group similar procedures and their associated variables to create an object-oriented design.
- *Optimizing a class diagram.* During the refactoring of an existing object-oriented application, one may want to reorganize the existing class structure in order to increase quality aspects such as maintainability or readability.

The relevance of this case study for TTC'16 stands out in that it presents a type of problem not presented in any contest before. In the case, the quality of the produced transformations is not strictly measured by the quality of the rules, but by their application orchestration, i.e., to execute the rules in the appropriate order and for the most appropriate matches to obtain high-quality output models. This implies to deal with a high computational complexity, since there are many

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

[1] All artifacts related to this case study can be found on GitHub at:
`https://github.com/martin-fleck/cra-ttc2016`

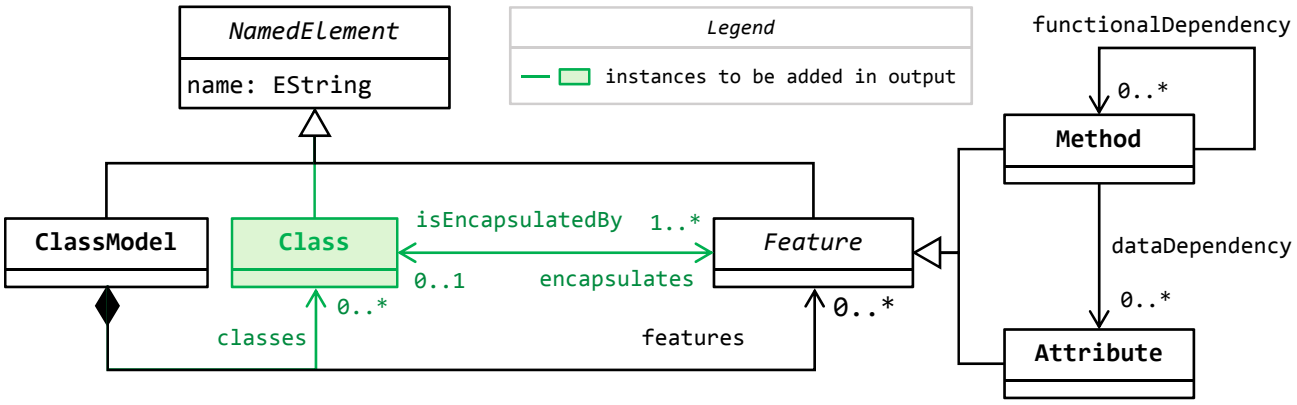


Figure 1: Class Model metamodel

different possibilities for orchestrating a set of rules. Indeed, we can categorize the CRA problem as a problem related to the partitioning of a set of labeled features (operations and attributes) into non-empty classes so that every feature is included in exactly one class. The number of possible partitions, i.e., classes, is given by the Bell number (cf. Equation 1). The nth of these numbers, B_n, counts the number of different ways a given set of n features can be divided into classes. If there are no features given (B_0), we can in theory produce exactly one partition (the empty set, $\emptyset$). The order of the classes as well as the order of the features within a class does not need to be considered as the semantic of a class diagram does not depend on that order.

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k \qquad (1)$$
$$B_0 = 1$$

Considering the first Bell numbers, which are shown below (cf. sequence A000110[2] in the OEIS online database for integer sequences), we can see that the number of partition possibilities grows exponentially and is already quite high for a low number of features. For example, an instance where you need to assign 15 features to an unknown amount of classes already yields 1382958545 different possibilities.

```
1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570, 4213597, 27644437,
190899322, 1382958545, 10480142147, 82864869804, 682076806159, 5832742205057,
51724158235372, 474869816156751, 4506715738447323, 44152005855084346,
445958869294805289, 4638590332229999353, 49631246523618756274, ...
```

In order to solve the case, as described in the next section, several techniques may be applied by the contestants.

2 Case Description

In this case study we propose a simplified version of the CRA problem. Contestants are given a set of methods and attributes as well as dependencies between them. Such a structure is also referred to as responsibilities dependency graph (RDG). Based on the RDG, the goal is to generate a high-quality class diagram (CD) model. The purpose is therefore to create a RDG2CD model transformation, where the RDG must evolve into a CD, categorized as an endogenous model transformation [MVG06], since both the input and output models conform to the same metamodel.

Figure 1 depicts the common metamodel that is used to represent both, the RDG and the output CD. The RDG is the subset of the metamodel containing only the features and their dependencies, and is represented in black, while the additional class and relationships needed to produce a CD are represented in green. The concepts depicted in the metamodel are summarized as follows:

Class Classes represent classes as known from object-oriented programming and modeling languages. A class hereby encapsulates certain functionality aspects in terms of methods, which in turn use data stored in attributes of instances

[2] http://oeis.org/A000110

Table 1: Summary of input models

	Input A	**Input B**	**Input C**	**Input D**	**Input E**
Attributes	5	10	20	40	80
Methods	4	8	15	40	80
Data Dep.	8	15	50	150	300
Functional Dep.	6	15	50	150	300

of the same or other classes. In this sense, classes serve as a container object for behavioral features (*methods*) and data features (*attributes*).

Method A method is used to describe a piece of executable behavior relating to a certain functionality. In the given metamodel, we abstract from the behavioral details, such as the sequence of performed actions, and focus on the dependencies generated through those actions. Specifically, we distinguish between *functional dependencies*, which refer to the dependencies among methods, and *data dependencies*, which refer to the dependencies between methods and attributes.

Attribute Attributes are used to store concrete data values. When used in an object-oriented language, all attributes of an object with their associated values represent the current state of the object. Changes of that state are typically executed through methods calls.

Encapsulation The encapsulation of classes and their features, i.e., methods and attributes, is given in the form of a bidirectional association (associations *isEncapsulatedBy* and *encapsulates*).

Functional Dependency A functional dependency represents a uni-directional relation from one method to another method, for example a method call. A functional dependency can therefore be defined as a relation $f \in M \times M$, where M is the set of all provided methods.

Data Dependency A data dependency represents a uni-directional relation from one method to an attribute, for example a read or write access. A data dependency can therefore be defined as a relation $d \in M \times A$, where M is the set of all provided methods and A is the set of all provided attributes.

2.1 Input

As input, contestants are given the above specified metamodel in Ecore as well as a set of input models conforming to the RDG subset. All input models are provided as XMI files and it can be assumed that the names of the methods and attributes contained in one RDG are unique. In total there are five input models with a varying degree of complexity. A summary of those models is given in Table 1.

2.2 Transformation

In this section we introduce the two main aspects that have to be considered by the contestants when producing output models.

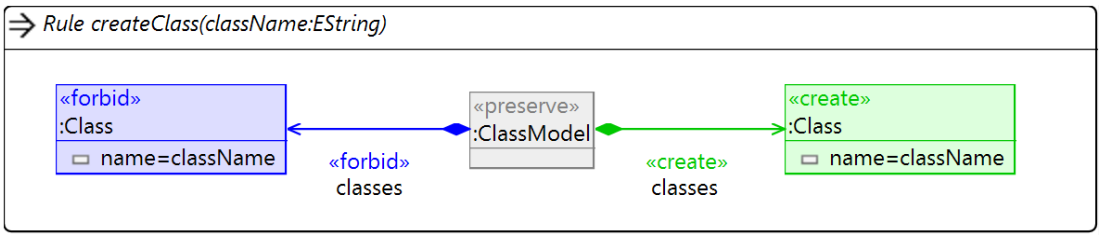
2.2.1 Transformation Rules.

As mentioned previously, the goal of this case study is to generate a high-quality CD model from the given RDG input model. To generate such a CD model, two major tasks have to be performed for each input model:

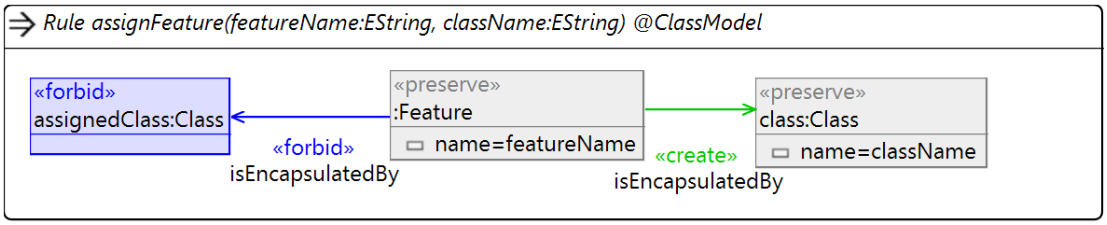
1. Create an appropriate amount of class instances, each having a unique name (the names do not have to be meaningful and can be assigned randomly).

2. Assign features to classes, i.e., set the encapsulation relationship between the classes and all the given features.

In order to create classes and assign features, two simple rules realizing the above mentioned tasks may be used. An example implementation of these rules in Henshin [ABJ$^+$10] is depicted in Figure 2.



(a) *createClass* rule.



(b) *assignFeature* rule.

Figure 2: Implementation of the rules in Henshin.

2.2.2 Transformation Goals.

The true challenge in this case study does not lie in the correct implementation of the transformation rules and creation of conforming CD models, but rather in the creation of high-quality CD models. Producing a class diagram where the right number of classes is chosen and a proper assignment of features is realized is a non-trivial task, and the contestants may realize it in several different ways.

To begin with, they may define more rules other than the two shown in Figure 2, or they can extend the given rules. Contestants also need to find the appropriate orchestration of the rules, with the appropriate input parameters, for producing high-quality class diagrams. Since this is quite complex and the state space can be huge (depending on the size of the input models), contestants may use several techniques for finding the rules orchestration, such as backtracking, state space exploration or dedicated search-based techniques.

Please note that we do not enforce any particular direction how the search for high-quality CD models is implemented. It may be done by exploiting current features of model transformation engines[3], e.g., used for formal analysis of models by using model checking techniques, by introducing new search features to the model transformation engines [AVS+14], by implementing search algorithms as transformations [DJVV14], combining search frameworks with model transformation engines [FTW15] or by translating the artifacts to dedicated encodings which already provide sophisticated search capabilities [EWZ14]. As several approaches for searching transformation spaces for good solutions have been proposed in the last years, we see this TTC case as an opportunity to evaluate and compare these diverse approaches based on a common example.

As for deciding on the quality of the obtained class diagrams, we use two common metrics for considering the quality of a grouping mechanism such as the grouping of functionality into classes: coupling and cohesion [BBL10]. *Coupling* refers to the number of external dependencies a specific group has, whereas *cohesion* refers to the dependencies within one group. Typically, low coupling is preferred as this indicates that a group covers separate functionality aspects of a system, improving the maintainability, readability and testability of the overall system [YC79]. On the contrary, the cohesion within one group should be maximized to ensure that it does not contain parts that are not part of its functionality. Mapping these definitions to our problem, we can calculate coupling and cohesion as the sum of external and internal dependencies, respectively.

One metric that combines coupling and cohesion into a single quality metric is the so-called *CRA-Index* [MJ14]. To be precise, the CRA-Index uses the coupling and cohesion ratios, i.e., the coupling and cohesion achieved considering the number of classes and attributes, and subtracts the former from the latter. Considering the meaning of coupling and cohesion as explained before, we can conclude that a higher CRA-Index relates to a higher quality of the class diagram. The formulae to calculate all necessary metrics and values are given below (taken from [MJ14]). Please note that $M(c)$ and $A(c)$ refer to all methods and attributes of class c, respectively.

[3] `https://wiki.eclipse.org/Henshin_State_Space_Tools`

$$CRA\text{-}Index = CohesionRation - CouplingRatio$$

$$CohesionRatio = \sum_{c_i \in Classes} \frac{MAI(c_i, c_i)}{|M(c_i)| \times |A(c_i)|} + \frac{MMI(c_i, c_i)}{|M(c_i)| \times |M(c_i) - 1|}$$

$$CouplingRatio = \sum_{\substack{c_i, c_j \in Classes \\ c_i \neq c_j}} \frac{MAI(c_i, c_j)}{|M(c_i)| \times |A(c_j)|} + \frac{MMI(c_i, c_j)}{|M(c_i)| \times |M(c_j) - 1|}$$

$$MMI(c_i, c_j) = \sum_{\substack{m_i \in M(c_i) \\ m_j \in M(c_j)}} DMM(m_i, m_j)$$

$$MAI(c_i, c_j) = \sum_{\substack{m_i \in M(c_i) \\ a_j \in A(c_j)}} DMA(m_i, a_j)$$

$$DMA(m_i, a_j) = \begin{cases} 1 & \text{if there is a dependency between method } m_i \text{ and attribute } a_j \\ 0 & \text{otherwise} \end{cases}$$

$$DMM(m_i, m_j) = \begin{cases} 1 & \text{if there is a dependency between method } m_i \text{ and } m_j \\ 0 & \text{otherwise} \end{cases}$$

In order to avoid division by zero in the calculation of the cohesion and coupling ratios, zero is assigned to the result of a division whenever its denominator is zero.

To sum up, the challenge of this case study is to find a way to properly orchestrate the given, extended, or newly created rules in order to optimize the quality of the produced class diagrams, which is achieved by optimizing the explained CRA-Index.

2.3 Output

In order to ensure validity of the generated output CD models, they must conform to the metamodel shown in Figure 1 and in addition must satisfy the following constraints:

- Every class must have a unique name, represented in OCL as:

```
-- unique class names
Class.allInstances()->isUnique(name)
```

- All features provided in the input model must be encapsulated by a class, represented in OCL as:

```
-- all features assigned
Feature.allInstances()->forAll(f | not f.isEncapsulatedBy.oclIsUndefined())
```

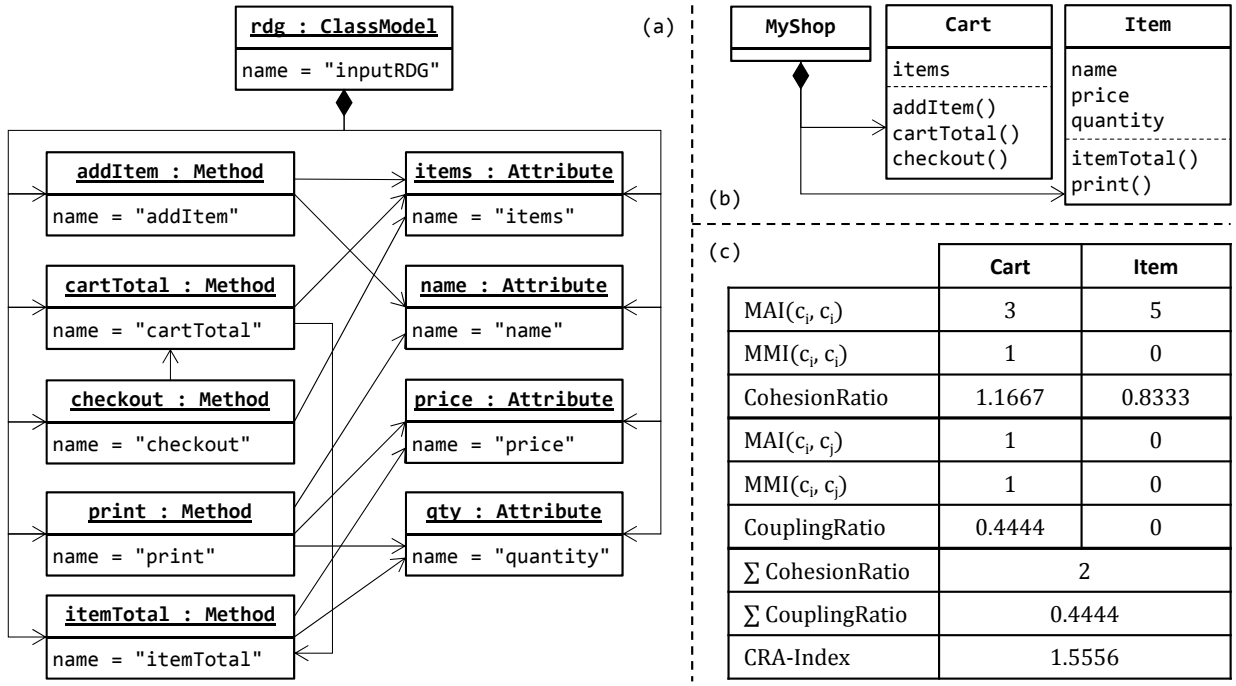
- There cannot be any empty classes. This is already enforced by a lower bound constraint in the metamodel.

Figure 3(a) depicts an example input RDG model composed of methods, attributes, and the relationships between them. Please note that in this figure we have not labelled the relationships for the sake of presentation. One possible (not necessarily optimal) output class diagram for this RDG is shown in Figure 3(b) in a simplified class diagram-like notation. The actual output CDs that need to be produced by the contestants must be in XMI format. Finally, the metrics obtained for the shown class diagram are depicted in the table in Figure 3(c).

3 Evaluation Criteria

For evaluating the provided solutions to the case study described in this paper, we are interested in the following five criteria. Both their description as well as the way to evaluate them can also be found in the provided evaluation spreadsheet.

Completeness & Correctness Completeness is a criteria indicating whether the provided solution/program always yields an output model as a result for the provided input models. On the other hand, correctness defines whether the generated output model conforms to the output metamodel and fulfils all constraints specified in Section 2.3. Specifically, correctness and completeness count how many of the five provided input models have been transformed into correct output models.



	Cart	Item
MAI(c_i, c_i)	3	5
MMI(c_i, c_i)	1	0
CohesionRatio	1.1667	0.8333
MAI(c_i, c_j)	1	0
MMI(c_i, c_j)	1	0
CouplingRatio	0.4444	0
$\sum$ CohesionRatio	2	
$\sum$ CouplingRatio	0.4444	
CRA-Index	1.5556	

Figure 3: Example input/output model pair with quality characteristics: (a) input model in abstract syntax, (b) output model in concrete syntax, (c) measures for output model.

Optimality With optimality the quality of the correctly generated models is evaluated, i.e., the CRA-Index of the output model. The higher the CRA-Index the better the quality of the output model. Here, the reviewers need to rank the solutions in relation to the other solutions provided and give them points on a scale between 1 and 10, where 1 refers to the worst (possible) solution and 10 refers to the best solution. To support this ranking, we provide the CRA-Index of our reference solution in the evaluation spreadsheet and a program that calculates the CRA-Index for a given class diagram.

Complexity With complexity we measure the efforts needed to provide search capabilities for good solutions as well as to evaluate the solutions based on the given metrics. For instance, this involves to evaluate how much effort has been invested to augment the provided rules, develop orchestration specifications such as providing an explicit control flow for the rules, implement search algorithms as transformations, or to implement transformations to dedicated encodings used for performing the search and back. Here, again, the reviewers need to rank the solutions in relation to the other solutions provided and give them points on a scale between 1 and 10, where 1 refers to the worst (possible) solution and 10 refers to the best solution.

Flexibility Flexibility measures how easy it is to modify the given solution to support additional/other quality metrics besides coupling, cohesion and the CRA-Index. For this criteria, reviewers need to estimate the effort it takes to integrate new objectives (such as fixing the number of classes to a given value) and give the provided solutions points on a scale between 1 and 10, where 1 refers to the worst (possible) solution, i.e., the solution where the most effort is needed, and 10 refers to the best solution, i.e., the integration can be done quickly.

Performance The performance evaluation consists of the measured execution time, i.e., the time it takes the provided solution to generate a high-quality output model for a given input model. Please note that reading the input model and writing the output model is not considered to be part of this performance evaluation. For Java-based solutions, we suggest using Java's internal time measurements, i.e., the method `java.lang.System.nanoTime()`, which is also used by the Apache Commons Lang's[4] `StopWatch` class. All performance values must be given exact to the millisecond, e.g., `03:02.426` meaning 3 minutes, 2 seconds and 426 milliseconds or in total `182426` milliseconds.

All criteria, except the complexity and flexibility of the solution, are evaluated separately on all provided input models.

[4] `https://commons.apache.org/proper/commons-lang/`

Table 2: Evaluation Schema

Criteria	Weight	MaxPoints	Total
Completeness & Correctness	1	10	10
Optimality	3	10	30
Complexity	2	10	20
Flexibility	2	10	20
Performance	2	10	20
Total			100

Depending on the criteria, each input model is given a specific weight that relates to the complexity of the model. Furthermore, the criteria are also weighted according to their importance. The maximum points and the weight of each criteria is specified in Table 2.

Using this table, which is incorporated in the provided spreadsheet, the result value, i.e., the *final score* of all provided solutions can be calculated. Each solution will receive a score between 0 and 100 and can therefore be ranked in comparison with other solutions.

4 Test artifacts

As a main test artefact, we provide an Excel sheet which needs to be filled out by the reviewers of the solutions. To support the reviewer in this task we provide a small Java application that checks the correctness and completeness of the provided output models and calculates the CRA-Index. To execute the application, we need to call the program with the respective class model:

```
java -jar CRAIndexCalculator.jar <xmi-model>
```

An example of the output of the application is depicted in Listing 1 and Listing 2.

Listing 1: Example output when providing a valid class model

```
java -jar CRAIndexCalculator.jar Cart_Item_cd.xmi

-----------------------------------------------
General Info
-----------------------------------------------
|Classes|    = 2
|Methods|    = 5
|Attributes| = 4
-----------------------------------------------

-----------------------------------------------
Correctness
-----------------------------------------------
All classes have different names: ok
All features are encapsulated in a class: ok
Correctness: ok
-----------------------------------------------

-----------------------------------------------
Optimality
-----------------------------------------------
The aggregated cohesion ratio is: 2.0
The aggregated coupling ratio is: 0.4444
This makes a CRA-Index of: 1.5555
-----------------------------------------------
```

Listing 2: Example output when providing an RDG model

```
java -jar CRAIndexCalculator.jar Cart_Item_rdg.xmi

-------------------------------------------
General Info
-------------------------------------------
|Classes|    = 0
|Methods|    = 5
|Attributes| = 4
-------------------------------------------

-------------------------------------------
Correctness
-------------------------------------------
All classes have different names: ok
Constraint violated! Feature addItem is not encapsulated in a class
Constraint violated! Feature cartTotal is not encapsulated in a class
Constraint violated! Feature checkOut is not encapsulated in a class
Constraint violated! Feature print is not encapsulated in a class
Constraint violated! Feature itemTotal is not encapsulated in a class
Constraint violated! Feature items is not encapsulated in a class
Constraint violated! Feature name is not encapsulated in a class
Constraint violated! Feature price is not encapsulated in a class
Constraint violated! Feature qty is not encapsulated in a class
Correctness: Violations found.
-------------------------------------------

-------------------------------------------
Optimality
-------------------------------------------
The aggregated cohesion ratio is: 0.0
The aggregated coupling ratio is: 0.0
This makes a CRA-Index of: 0.0
-------------------------------------------
```

References

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Proceedings of 13th International Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010.

[AVS+14] Hani Abdeen, Dániel Varró, Houari A. Sahraoui, András Szabolcs Nagy, Csaba Debreceni, Ábel Hegedüs, and Ákos Horváth. Multi-objective optimization in rule-based design space exploration. In *Proceedings of the 29th International Conference on Automated Software Engineering (ASE'14)*, pages 289–300. IEEE/ACM, 2014.

[BBL10] M. Bowman, L.C. Briand, and Y. Labiche. Solving the Class Responsibility Assignment Problem in Object-Oriented Analysis with Multi-Objective Genetic Algorithms. *IEEE Transactions on Software Engineering*, 36(6):817–837, 2010.

[DJVV14] Joachim Denil, Maris Jukss, Clark Verbrugge, and Hans Vangheluwe. Search-Based Model Optimization Using Model Transformations. In *Proceedings of the 8th International Conference on System Analysis and Modeling (SAM'14)*, volume 8769 of *LNCS*, pages 80–95. Springer, 2014.

[EWZ14] Dionysios Efstathiou, James R. Williams, and Steffen Zschaler. Crepe complete: Multi-objective optimisation for your models. In *Proceedings of the 1st International Workshop on Combining Modelling with Search- and Example-Based Approaches (CMSEBA'14) @ MODELS*, volume 1340, pages 25–34. CEUR-WS.org, 2014.

[FTW15] Martin Fleck, Javier Troya, and Manuel Wimmer. Marrying Search-based Optimization and Model Transformation Technology. In *Proceedings of the 1st North American Search Based Software Engineering Symposium (NasBASE'15)*, 2015.

[MJ14] Hamid Masoud and Saeed Jalili. A clustering-based model for class responsibility assignment problem in object-oriented analysis. *Journal of Systems and Software*, 93(0):110 – 131, 2014.

[MVG06] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[YC79] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., 1st edition, 1979.

Solving the Class Responsibility Assignment Case with UML-RSDS

K. Lano, S. Yassipour-Tehrani, S. Kolahdouz-Rahimi
Dept. of Informatics, King's College London, Strand, London, UK;
Dept. of Software Engineering, University of Isfahan, Isfahan, Iran

Abstract

This paper provides a solution to the class responsibility assignment case using UML-RSDS. We show how search-based software engineering techniques can be combined with traditional MT techniques to handle large search spaces.

Keywords: Class responsibility assignment; Search-based software engineering; UML-RSDS.

1 Introduction

This case study [1] is an endogenous transformation which aims to optimally assign attributes and methods to classes to improve a measure, class-responsibility assignment (CRA-index), of class diagram quality. We provide a specification of the transformation in the UML-RSDS language [3, 4] using search-based software engineering techniques (SBSE) [2].

UML-RSDS is a model-based development language and toolset, which specifies systems in a platform-independent manner, and provides automated code generation from these specifications to executable implementations (in Java, C$^\#$ and C++). Tools for analysis and verification are also provided. Specifications are expressed using the UML 2 standard language: class diagrams define data, use cases define the top-level services or functions of the system, and operations can be used to define detailed functionality. Expressions, constraints, pre and postconditions and invariants all use the standard OCL 2.4 notation of UML 2.

For model transformations, the class diagram expresses the metamodels of the source and target models, and auxiliary data and functionalities can also be defined. Use cases define the transformations and their subtransformations: each use case has a set of pre and postconditions which define its intended functionality. The postconditions act as transformation rules. A use case can include others, and may have an activity to define its detailed behaviour.

2 Class responsibility assignment

In our solution, we combine the SBSE technique of genetic algorithms with a traditional endogenous model transformation. This is a particular case of a general strategy used in UML-RSDS to combine SBSE and MT (Figure 1), where transformations are used to pre- and post-process the input data and results of an evolutionary algorithm. We have selected a genetic algorithm (GA) for SBSE because the CRA problem is akin to scheduling and bin-packing problems, for which genetic algorithms have proved widely successful. We observed that the problem seems to satisfy the property of possessing 'building blocks' – in this case groups consisting of a method plus a group of attributes which it depends upon and no other method does. Such groups must always be placed in the same class and hence form a higher granularity unit (compared to individual features) from which potential solutions can be composed.

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

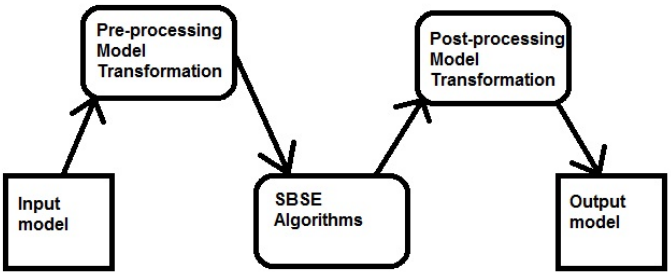


Figure 1: Integration of model transformations and SBSE

The first part of the solution is an endogenous transformation (Figure 2) which (i) identifies the building blocks and places these in separate classes: the *createClasses* transformation in Figure 2; (ii) refactors the class model to reduce coupling: the *refactor* transformation; (iii) removes empty classes: the *cleanup* transformation. Finally, the *measures* transformation displays the CRA-index and other measures of the intermediate solution. These transformations are co-ordinated by the *preprocess* transformation. The metamodel is produced from the Ecore metamodel provided.

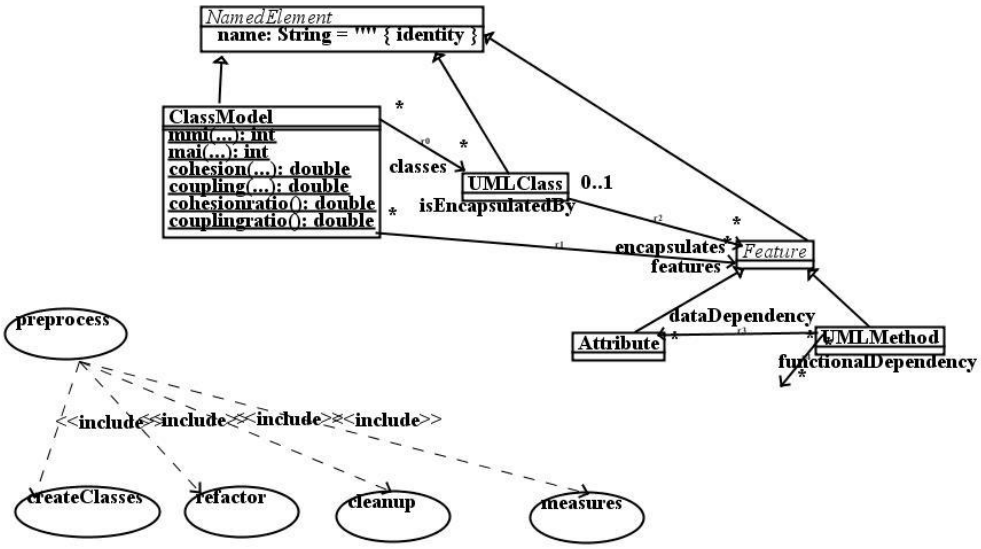


Figure 2: Pre-transformation (based on architectureCRA.ecore)

The rules (postconditions) for *createClasses* include creation of a default class, which contains all the methods that are not linked to any other feature (their sets of functional and data dependencies are empty):

```
UMLMethod::
  dataDependency.size = 0 & functionalDependency.size = 0   =>
      UMLClass->exists( c | c.name = "Class0" & self : c.encapsulates )
```

and a rule to place a method (*self*) into any class *c* which contains all of the attributes it depends on:

```
UMLMethod::
  dataDependency.size > 0 & c : UMLClass &
  dataDependency <: c.encapsulates@pre  =>  self : c.encapsulates
```

<: denotes the subset relation. Other rules create classes for each 'chunk' of a method plus the attributes which are exclusively or primarily depended on by that method.

The *refactor* transformation moves methods m and attributes a from one class *self* to an alternative class c, if there are more dependencies linking the feature to c than to *self*. Eg., for methods we have:

```
UMLClass::
m : encapsulates@pre & m->oclIsKindOf(UMLMethod) & c : UMLClass &
depends = m.dataDependency->union(m.functionalDependency) &
depends->intersection(c.encapsulates@pre)->size() >
      depends->intersection(encapsulates@pre)->size()  =>  m : c.encapsulates
```

Although this transformation may decrease the CRA-index, it generally reduces coupling.

Finally, *cleanup* deletes empty classes:

```
UMLClass::
  encapsulates.size = 0   =>  self->isDeleted()
```

The *preprocess* transformation co-ordinates the other transformations. It has no rules of its own, but it has an activity to control the subordinate transformations, which *preprocess* accesses via $\ll$ *include* $\gg$ dependencies:

```
while Feature->exists(isEncapsulatedBy.size = 0)
do execute ( createClasses() )  ;
while UMLClass->exists( c | c.name /= "Class0" & c.encapsulates.size = 1 )
do execute ( refactor() )  ;
execute ( cleanup() ) ;
execute ( measures() )
```

createClasses is iterated until all features are encapsulated in some class, then *refactor* is iterated until all normal classes have at least 2 features.

3 Genetic algorithm

A general GA specification is provided in the UML-RSDS libraries (Figure 3). This can be reused and adapted for specific problems, by providing a problem-specific definition of the *fitness* function, the content of *GATrait* items and values, and the functions determining which individuals survive, reproduce or mutate from one generation to the next. *Math* is an external class providing access to the Java *random*() method.

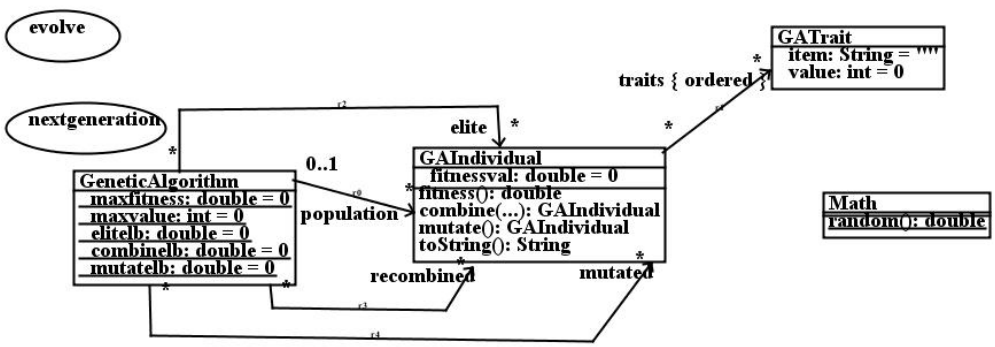


Figure 3: Genetic algorithm specification in UML-RSDS

The general definitions of *evolve* and *nextgeneration* are as follows:

```
Use Case, name: evolve

GeneticAlgorithm::
p : population@pre & GeneticAlgorithm.isUnfit(p)  =>  p->isDeleted()

GeneticAlgorithm::
p : population & GeneticAlgorithm.isElite(p)   =>   p : elite

GeneticAlgorithm::
```

```
p : elite & q : population &
q.fitnessval < p.fitnessval & GeneticAlgorithm.isCombinable(p,q) =>
                                        p.combine(q) : recombined

GeneticAlgorithm::
p : population & GeneticAlgorithm.isMutatable(p)  =>  p.mutate() : mutated


Use Case, name: nextgeneration

GeneticAlgorithm::
true =>
     population = elite@pre->union(recombined@pre)->union(mutated@pre)

GeneticAlgorithm::
true =>
     elite = Set{} & recombined = Set{} & mutated = Set{}

GeneticAlgorithm::
p : population  =>  p.fitnessval = p.fitness()

GeneticAlgorithm::
population.size > 0  =>
     GeneticAlgorithm.maxfitness = population->collect(fitnessval)->max()
```

evolve deletes unfit individuals (eg., those with fitness below the median fitness level), preserves the best individuals (those in the best 20%), and mutates and recombines individuals that meet the necessary criteria. Mutation consists of incrementing or decrementing the *value* of a randomly selected trait. Crossover takes place at a randomly-selected trait position. *nextgeneration* assembles the next population, and recalculates fitness for all individuals, and identifies the top fitness value.

To model the CRA problem in a GA, individuals represent possible assignments of features to classes, and have traits t for each feature f in the model, and *item* is the name of the feature. The trait *value* is the index number of the class to which the feature is assigned, ie.:

$$t.value = UMLClass.allInstances \rightarrow indexOf(f.isEncapsulatedBy.any)$$

The fitness value is the CRA-index, computed using the definitions of [1]. This approach means that each individual has a trait for every feature in the model, and this makes processing very slow for substantial models. For larger instances of the CRA problem, a more compact representation of models as individuals would be necessary, for example, by grouping features into chunks and defining one trait per chunk.

The functions *mmi* and *mai* of [1] are defined as follows:

```
ClassModel::
query static mmi(ci : UMLClass, cj : UMLClass) : int
pre: true
post:
  result = ci.encapsulates->select( mi |
       mi : UMLMethod )->collect( m |
            cj.encapsulates->intersection(m.functionalDependency)->size() )->sum()


ClassModel::
query static mai(ci : UMLClass, cj : UMLClass) : int
pre: true
post:
  result = ci.encapsulates->select( mi |
       mi : UMLMethod )->collect( m |
           cj.encapsulates->intersection(m.dataDependency)->size() )->sum()
```

Likewise, the coupling and cohesion ratios can be defined in OCL.

The *ga*(*iter* : *int*) use case performs *initialise* to initialise the population with 50 copies of the model produced by *preprocess*, and 100 random individuals, then it iterates *evolve* and *nextgeneration* for *iter* times.

In a final phase, an optimal individual *gmax* produced by the genetic algorithm is mapped to a class model by the *postprocess* use case:

```
GeneticAlgorithm::
population.size > 0 & gmax = population->selectMaximals(fitnessval)->any() =>
  gmax.traits->forAll( t |
      UMLClass.allInstances->at(t.value) : Feature[t.item].isEncapsulatedBy )
```

$E[str]$ is the instance of entity type E with primary key value *str*. Following this, *cleanup* may be needed to remove any empty classes.

4 Results

Table 1 gives some typical results for the five example models. We show the execution times for *preprocess*, *ga* and *postprocess* separately. For test A, *createClasses* results in 4 classes, with 3, 2, 2 and 2 features respectively, cohesion ratio 4 and coupling ratio 1. Applying *refactor* reduces the model to 2 classes, with higher average cohesion, and a lower coupling ratio (0.5). The CRA is 1.6667. Applying the genetic algorithm to this recovers the first solution with CRA 3, after 10 generations. For test B, applying *createClasses* produces a solution with 9 classes and CRA 2.5, but with 3 classes containing only 1 feature each. Applying *refactor* improves the cohesion ratio and eliminates the 'orphen' features, but reduces the CRA to -1.5 (7 classes). Applying the GA for 10 generations produces an improved model with CRA 2.75. For model C, *createClasses* yields an initial model with CRA -3.917, *refactor* reduces this to -4.09, but applying the GA for 15 generations improves this to 0.6175. For model D, the respective values are -20.83, -2.807, 0.744, and for model E -44, -20.37, -8.1.

Test	*CRA after createClasses*	*CRA after refactor*	*CRA after* GA	*Execution time* (preprocess + ga + postprocess)
A	3	1.666	3	15ms + 47ms + 0ms
B	2.5	-1.5	2.75	32ms + 1s + 7ms
C	-3.917	-4.09	0.6175	77ms + 42s + 11ms
D	-20.83	-2.807	0.744	500ms + 265s + 99ms
E	-44	-20.37	-8.1	2012ms + 2322s + 219ms

Table 1: Example test results

All solution artifacts are available at www.dcs.kcl.ac.uk/staff/kcl/umlcra. The specification is in the file mm.txt, and the use cases and operations are listed in crausecases.txt and craoperations.txt. Solution models are in cresult.xmi, dresult.xmi, etc.

Conclusions

We have shown that a general-purpose genetic algorithm can be used in combination with MT to obtain good results for the CRA problem. General recombination and mutation strategies are used, for example, mutation simply moves one feature from one class to another. Improved results could probably be produced if specialised operators were used, eg., moving a method would also require moving the attributes that it exclusively depends upon. A significant aspect of our approach is that we have specified the GA and MT system components in the same formalism (UML and OCL), rather than using hetrogeneous technologies.

References

[1] M. Fleck, J. Troya, M. Wimmer, *The Class Responsibility Assignment Case*, TTC 2016.

[2] M. Harman, *Search-based software engineering*, ICCS 2006, LNCS vol. 3994, Springer-Verlag, pp. 740–747, 2006.

[3] K. Lano, S. Kolahdouz-Rahimi, *Constraint-based specification of model transformations*, Journal of Systems and Software, vol. 88, no. 2, February 2013, pp. 412–436.

[4] The UML-RSDS toolset and manual, `http://www.dcs.kcl.ac.uk/staff/kcl/uml2web/umlrsds.pdf`, 2016.

An NMF solution to the Class Responsibility Assignment Case

Georg Hinkel
FZI Research Center of Information Technologies
Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
hinkel@fzi.de

Abstract

This paper presents a solution to the Class Responsibility Assignment (CRA) case at the Transformation Tool Contest (TTC) 2016 using the .NET Modeling Framework (NMF). The goal of this case was to find a class model with high cohesion but low coupling for a given set of attributes and methods with data dependencies and functional dependencies. The degree in which a given class model fulfills these properties is measured through the CRA-Index. We propose a general-purpose code solution and discuss how this solution can benefit from incrementality. In particular, we show what steps are necessary to create an incremental solution using NMF Expressions and discuss its performance.

1 Introduction

The Class Responsibilities Assignment (CRA) problem is a basic problem in an early stage of software design. Usually, it is solved manually based on experience, but early attempts exist to automate the solution of this problem through multi-objective search [BBL10]. Given the exponential size of the search space, the problem cannot be solved by brute force. Instead, often genetic search algorithms are applied.

The CRA problem has been formulated as a case for the Transformation Tool Contest 2016[1]. This paper contains a solution to this particular case description.

The advantage of approaches such as genetic search algorithms or simulated annealing is that they can find a good solution even when the fitness function is not well understood. In the case of the CRA, however, the fitness function is relatively simple. Therefore, we refrained from using these tools, as we think they cannot unveil their potential in this case. Further, the cost of generality often is a bad performance, which may make such an approach not suitable for large input models, for example when a large system design should be reconsidered. Therefore, we created a fully custom solution using general-purpose code without the background of a framework. We are interested to see how we compare to search tools based on genetic algorithms in this case.

Furthermore, we detected that a lot of computational effort is done repeatedly in our solution. This yields a potential of further performance improvements through the introduction of memoization and incrementality, i.e., insertion of buffers that are managed by the evaluation system in order to avoid performing the same calculations multiple times when the underlying data has not changed in between.

The results show that our batch solution has a good performance, solving the largest provided input model within a few seconds and creating output models with a good CRA score. Further, the solution could be memoized

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

[1] `https://github.com/martin-fleck/cra-ttc2016/blob/master/case_description/TTC2016_CRA.pdf`

and even incrementalized with few changes to the code, showing the possibilities of our implicit incrementalization approach NMF Expressions. However, the performance results for the incremental version of the solution were discouraging as the largest model took almost one and a half minutes to complete.

Our solution is publicly available on GitHub[2] and SHARE[3].

The remainder of this paper is structured as follows: Section 2 gives a very brief overview on NMF. Section 3 presents our solution. Section 4 discusses the potential of incremental execution for our solution. Section 5 evaluates our approach before Section 6 concludes the paper.

2 The .NET Modeling Framework

The .NET Modeling Framework (NMF) [Hin16] is a framework designed to support model-driven engineering on the .NET platform. It allows users to generate code for Ecore metamodels and load and save EMF models in most cases (i.e., when the Ecore metamodel refrains from using generic types, factories or custom XML handlers). For this, a given Ecore metamodel is transformed into NMF's own meta-metamodel NMeta for which code can be generated.

Besides this, NMF contains the implicit incrementalization system NMF Expressions which allows developers of model analyses to run their analyses incrementally without changing the code. This means that incremental execution can be implemented at practically no cost and without degrading understandability or maintainability of the analysis as almost no changes have to be performed.

3 The Class Responsibilities Assignment Case Solution

The NMF solution to the CRA case is divided into four parts: Loading the model, creating an initial correct and complete solution, optimization and serializing the resulting model. Therefore, the optimization is entirely done in memory. We first give an overview on the solution concept and then describe the steps in more detail.

3.1 Overview

The general idea of our solution is to create an initial complete and correct model which is incrementally improved in a greedy manner until no more improvements can be found. For this, we apply a bottom-up strategy and start with a class model where each feature is encapsulated in its own class and gradually merge these classes until no improvement can be found. Here, we risk that we may get stuck in a local maximum of the CRA-index. The solution could be further extended to apply probabilistic methods such as simulated annealing to overcome local maxima, but the results we achieved using the greedy algorithm were quite satisfactory and we therefore did not take any further step in this direction.

The idea of the optimization is to keep a list of possible class merges and sort them by the effect that this merge operation has to the CRA index. We then keep applying the most promising merge as long as this effect is positive. Here, merging two classes c_i and c_j means to create a new class c_{ij} that contains the features of both classes. The new class is then added to the model while the old classes are removed.

Using $MAI(c_i, c_j)$ as the relation counting data dependencies between c_i and c_j, we observe that

$$MAI(c_{ij}, c_{ij}) = MAI(c_i, c_i) + MAI(c_i, c_j) + MAI(c_j, c_i) + MAI(c_j, c_j)$$

and likewise for MMI that counts method dependencies. Then, the difference in cohesion $\Delta Coh(c_i, c_j)$ when merging c_i and c_j to c_{ij} can be expressed as follows:

$$\begin{aligned}\Delta Coh(c_i, c_j) = &\frac{MAI(c_i, c_i) + MAI(c_i, c_j) + MAI(c_j, c_i) + MAI(c_j, c_j)}{(|M_i| + |M_j|)(|A_i| + |A_j|)} - \frac{MAI(c_i, c_i)}{|M_i||A_i|} - \frac{MAI(c_j, c_j)}{|M_j||A_j|} \\ &+ \frac{MMI(c_i, c_i) + MMI(c_i, c_j) + MMI(c_j, c_i) + MMI(c_j, c_j)}{(|M_i| + |M_j|)(|M_i| + |M_j| - 1)} - \frac{MMI(c_i, c_i)}{|M_i|(|M_i| - 1)} - \frac{MMI(c_j, c_j)}{|M_j|(|M_j| - 1)}.\end{aligned}$$

The effect that this merge operation has on the coupling is more complex and requires analyzing what other classes are affected when merging c_i and c_j, i.e., which classes use or are used by a feature from either c_i or c_j. The effect on the coupling $\Delta Cou(c_i, c_j)$ between c_i and c_j can be expressed as

$$\Delta Cou(c_i, c_j) = -\frac{MAI(c_i, c_j)}{|M_i||A_j|} - \frac{MAI(c_j, c_i)}{|M_j||A_i|} - \frac{MMI(c_i, c_j)}{|M_i|(|M_j| - 1)} - \frac{MMI(c_j, c_i)}{|M_j|(|M_i| - 1)}.$$

[2] `http://github.com/georghinkel/TTC2016CRA`

[3] `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC16_NMF.vdi`

We then calculate the effect $\Delta CRA(c_i, c_j)$ of merging classes c_i and c_j simply as $\Delta Coh(c_i, c_j) - \Delta Cou(c_i, c_j)$.

Using this heuristic, we do not need to compute the CRA metric every time we perform merges of two classes. Instead, our heuristic is an estimate for the derivation of the fitness function when a merge operation is performed.

3.2 Loading the Model

Loading a model in NMF is very straight-forward. Once the code for the metamodel is generated, we create a new repository, resolve the file path of the input model into this respository and cast the single root element of this model as a `ClassModel`. If the model contains cross-references to other models, these models are loaded automatically into the same repository. Loading the model is depicted in Listing 1.

```
1 var repository = new ModelRepository();
2 var model = repository.Resolve(args[0]);
3 var classModel = model.RootElements[0] as ClassModel;
```

Listing 1: Loading the model

For this to work, only a single line of code in the assembly metadata is necessary to register the metamodel attached to the assembly as an embedded resource. This automatically registers the model classes with the serializer such that we do not have to activate the package or anything in that direction.

3.3 Optimization

To conveniently specify the optimization, we first specify some inline helper methods to compute the MAI and MMI of two classes. The sums in the definition of MAI and MMI can be specified conveniently through the query syntax of C#. The implementation for MAI is depicted in Listing 2, the implementation of MMI is equivalent.

```
1 var MAI = new Func<IClass, IClass, double>((cl_i, cl_j) =>
2     cl_i.Encapsulates.OfType<Method>().Sum(m => m.DataDependency.Intersect(cl_j.Encapsulates).Count()));
```

Listing 2: Helper function for MAI

With the help of these functions, we generate the set of merge candidates, basically by generating the cross product of classes currently in the class model. To do this, we need to find the classes for which the MAI and MMI will have an effect when c_i and c_j are merged. These can be obtained through the query depicted in Listing 3 where the opposite reference for the data dependency property is precalculated as it does not change during the optimization.

```
1 var dataDependingClasses =
2    (from att in classIAttributes.Concat(classJAttributes)
3     from meth in dataDependencies[att]
4     select meth.IsEncapsulatedBy).Distinct();
5 var dataDependentClasses =
6    (from meth in classIMethods.Concat(classJMethods)
7     from dataDep in meth.DataDependency
8     select dataDep.IsEncapsulatedBy).Distinct();
```

Listing 3: Computing affected classes

Similar queries detect the affected classes for coupling based on functional dependencies between methods.

To rate the candidates for merge operations, we assign an effect to them, which is exactly our aforementioned $\Delta CRA(c_i, c_j)$. The implementation is depicted in Listing 4.

```
1 var prioritizedMerges = (from cl_i in classModel.Classes where cl_i.Encapsulates.All(f => f is IAttribute)
2    from cl_j in classModel.Classes where cl_i.Name.CompareTo(cl_j.Name) < 0
3    select new { Cl_i = cl_i, Cl_j = cl_j, Effect = Effect(cl_i, cl_j)})
4    .OrderByDescending(m => m.Effect);
```

Listing 4: Sorting the merges by effect

We sort the possible merges and perform the most promising merge. We make use of the lazy evaluation of queries in .NET, which means that the creation of tuples, assigning effects and sorting is performed each time we access the query results. We do this repeatedly until the most promising merge candidate has an estimated effect ΔCRA below zero.

However, there is a potentially counter-intuitive issue here. The problem is that NMF takes composite references very seriously, so deleting a model element from its container effectively deletes the model element.

This in turn means that each reference to this model element is reset (to prevent the model pointing to a deleted element). This includes the reference *encapsulatedBy* and therefore also its opposite *encapsulates*, which means that as soon as we remove a class from the container, it immediately loses all of its features. Therefore, before we can delete the classes c_i and c_j, we need to store the features in a list and then add them to the newly generated class (cf. Listing 5).

```
1 var newFeatures = nextMerge.Merge.Cl_i.Encapsulates.Concat(nextMerge.Merge.Cl_j.Encapsulates).ToList();
2 classModel.Classes.Remove(nextMerge.Merge.Cl_i);
3 classModel.Classes.Remove(nextMerge.Merge.Cl_j);
4 var newClass = new Class() { Name = "C" + (classCounter++).ToString() };
5 newClass.Encapsulates.AddRange(newFeatures);
```

Listing 5: Performing merge operations

We run the code block in Listing 5 first setting `allClasses` to `false` in order to prioritize classes that only contain attributes and then repeat this procedure for all classes due to obtain better results.

3.4 Serializing the resulting model

NMF requires an identifier attribute of a class to have a value, in case the model element is referenced elsewhere. Therefore, we need to give a random name to the class model.

```
1 classModel.Name = "Optimized␣Class␣Model";
2 repository.Save(classModel, Path.ChangeExtension(args[0], ".Output.xmi"));
```

Listing 6: Saving the resulting model

Afterwards, the model can be saved simply by telling the repository to do so. This is depicted in Listing 6.

4 Memoization and Incrementalization

The heart and soul of our solution is to repeatedly query the model for candidates to merge classes and rank them according to our heuristic. Therefore, the performance of our solution critically depends on the performance to run this query. If the class model at a given point in time contains $|C|$ classes, this means that $|C|^2$ merge candidates must be processed, whereas only $2|C|$ merge candidates are removed and $|C|-2$ new merge candidates are created in the following merge step. Therefore, if we made sure we only process changes, we could change the quadratic number of classes to check to a linear one, improving performance.

Therefore, an idea to optimize the solution would be to maintain a balanced search tree with the heuristics for each candidate as key and dynamically update this tree when new merge candidates arise. The MAI and MMI of two classes does not change between subsequent calls and can be memoized.

However, we suggest that an explicit management of such a search tree would drastically degrade the understandability, conciseness and maintainability of our solution. In most cases, these quality attributes have a higher importance than performance since they are usually much tighter bound to cost, especially when performance is not a critical concern. Furthermore, it is not even clear whether the management of a search tree indeed brings advantages since the hidden implementation constant may easily outweigh the benefits of asymptotically better solutions.

This problem can be solved using implicit incrementalization approaches such as NMF Expressions [HH15]. Indeed, our solution has to be modified only at a few places and the resulting code can be run incrementally. Apart from namespace imports, developers only need to switch the function type `Func<>` to `MemoizedFunc<>` to enable memoization or `IncrementalFunc<>` to enable incremental execution.

5 Evaluation

The results achieved on an Intel Core i5-4300 CPU clocked at 2.49Ghz on a system with 12GB RAM are depicted in Table 1 for the solution in normal and memoized mode. Each measurement is repeated 5 times.

Furthermore, the performance figures indicate that in batch mode, at least for the smaller models, the optimization takes much less time than loading the model. Even for the largest provided reference models, the optimization completes in a few seconds. The incrementalized version was much slower than the normal or memoized execution and is therefore skipped in the scope of this paper.

We also depicted the rank of our (memoized) solution both in terms of performance and CRA-Index among all the solution submissions at the contest in their improved versions plus the reference solution. Our solution

	Input A	Input B	Input C	Input D	Input E
Correctness	•	•	•	•	•
Completeness	•	•	•	•	•
CRA-Index	3	3.08	1.63	-0.68	2.16
Model Deserialization	192ms	185ms	163ms	160ms	155ms
Optimization (normal)	10.4ms	18ms	96.8ms	1,422ms	11,295ms
Optimization (memoized)	18.5ms	20.2ms	63.4ms	700ms	6,877ms
Model Serialization	11ms	11ms	12ms	10ms	11ms
CRA-Index (rank)	1-10	6-7	6	6	3
Execution Time (rank)	1	1	1	1	1

Table 1: Summary of Results for memoization solution. Shared ranks in the CRA-values means that multiple solutions created solutions with the same CRA-index. There were 10 solutions in total.

was the fastest for all reference input models. For the larger models, the Excel solution[4] was eventually better since the employed Markov Clustering Algorithm has a better asymptotic complexity. However, the quality of the result models was not as good as other solutions that applied genetic search algorithms.

The solution consists of 227 lines of code (+12 for the incrementalized one), including imports, comments, blank lines and lines with only braces plus the generated code for the metamodel and one line of metamodel registration. Therefore, we think that the solution is quite good in terms of conciseness.

A downside of the solution is of course that it is very tightly bound to the CRA-index as fitness function and does not easily allow arbitrary other conditions to be implemented easily. The heuristic to first merge classes that only contain attributes also incorporates insights beyond the pure calculation of the metric. For example, to fix the number of classes, one would have to perform a case distinction whether the found optimal solution has more or less classes and then either insert empty classes or continue merging classes.

A comparison with other solutions demonstrated at the TTC contest has shown that our solution is faster than any other proposed solution, for some of them even by multiple orders of magnitude. However, the quality of the result models in terms of the CRA-index is not as good as for other solutions, though it is by far also not the worst. Thus, our solution may serve as a baseline for the advantages and disadvantages of more elaborate search tools, for example based on genetic search.

6 Conclusion

In this paper, we presented our solution to the CRA case at the TTC 2016. The solution shows the potential of simple derivation heuristics. The results in terms of performance were quite encouraging. We also identified a good potential of incrementality in our solution. We were able to apply incrementality by changing just a few lines of code. However, the resulting solution using an incremental query at its heart is much slower, indicating that the overhead implied by managing the dynamic dependency graph in our current implementation still outweighs the savings because of the improved asymptotical complexity. We are working on the performance of our incrementalization approach and will use the present CRA case as a benchmark.

References

[BBL10] Michael Bowman, Lionel C Briand, and Yvan Labiche. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. *Software Engineering, IEEE Transactions on*, 36(6):817–837, 2010.

[HH15] Georg Hinkel and Lucia Happe. An NMF Solution to the TTC Train Benchmark Case. In Louis Rose, Tassilo Horn, and Filip Krikava, editors, *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences*, volume 1524 of *CEUR Workshop Proceedings*, pages 142–146. CEUR-WS.org, July 2015.

[Hin16] Georg Hinkel. NMF: A Modeling Framework for the .NET Platform. Technical report, Karlsruhe, 2016.

[4] `http://www.transformation-tool-contest.eu/solutions/cra/TTC_2016_paper_6.pdf`

A heuristic approach for resolving the Class Responsibility Assignment Case

Maximiliano Vela
maximiliano.vela@hotmail.com

Yngve Lamo
yla@hib.no

Fazle Rabbi
fra@hib.no

Fernando Macías
fmac@hib.no

Bergen University College
Bergen, Norway

Abstract

This paper describes one of the solutions for the ninth Transformation Tool Contest (TTC '16)[1], which resolves the Class Responsibility Assignment Case using a transformation tool based on Microsoft Excel and Visual Basic. In this project, these relatively unusual technologies are used to effectively enhance the processing of large models and matrices throughout different test cases proposed for the competition. Given a Responsibility Dependency Graph, the solution analyzes the relationships among the given features and produces a class diagram that aims to get the highest possible cohesion and lowest possible coupling. The solution is based on an adaptation of the Markov Clustering Algorithm used to manage bi-directional, un-weighted sets of nodes and grouping them into clusters. The aim of this work is to take one step further in the graph/model optimization field through the treatment of matrices, a strategy that is not so widely used.

1 Introduction

Cohesion is a property that refers to the degree to which the elements of a module belong together, while coupling is the degree of interdependence between different modules. In order to generate high-quality models, the need for measuring and comparing how strong the relationship is between different components of a module is extremely important. Even though in some cases the coupling will always be inevitably higher than the cohesion, there will always exist ways to optimize the resulting model as much as possible.

The algorithm used in this solution to generate the resulting class diagram is based on the foundations of the Markov Clustering Algorithm [SVD00] [KM09]. However, most of the steps have been slightly or completely altered since they have been deemed not as effective for the proposed problem.

As described in the problem definition, for a total of 18 input features there are 682.076.806.159 possible class diagrams built upon different feature combinations, so the time it takes for the solution to generate a potentially optimal output is essential.

The objective of the proposed solution[2] is to perform a model transformation from the initial Responsibility

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

[1] The full problem description can be found at: github.com/martin-fleck/cra-ttc2016/

[2] The proposed solution and other useful files can be found at: github.com/maximiliano-vela/ttc-2016 - SHARE demonstration/reviewing VM: XP-TUe_XP_Excel_v2.vdi @ http://is.ieis.tue.nl/staff/pvgorp/share/

Dependency Graph received as input into a Class Diagram which attempts to achieve the highest possible CRA-Index by having as high cohesion and as low coupling as possible.

The meta-model transformation relies in linking each feature (method or attribute) to a class that encapsulates it, as shown in Figure 1.

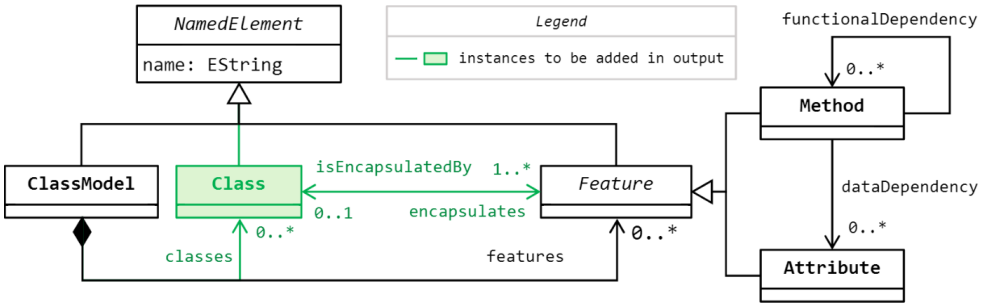


Figure 1: Responsibility Dependency Graph meta-model[3]

2 Motivation and Features

At the beginning of this project, the idea of using Microsoft Excel as the main part of the transformation was merely a result of the developer's working experience with Visual Basic in different fields of the industry. However, since the moment we found out about the Markov Clustering Algorithm and its use within matrix/graph operations, Excel naturally started to sound more and more convenient. Additionally, since matrices are worked upon the actual spreadsheet, the debugging of the algorithm as well as further experimentation with it became significantly easier. The tool is highly transparent, very easy to use, requires little to no technology stack at all (for Windows users, that is) and its transformation process, described in the following section, is relatively simple.

3 Algorithm Implementation

The implementation starts off with a pre-processing step that transforms the input file into meaningful data that will shortly after be used in order to generate the classes. The results are finally formatted as an .xmi file for further use. Several inputs have been tested for the competition, however Input A (the smallest one) will be the one used as an example in this section.

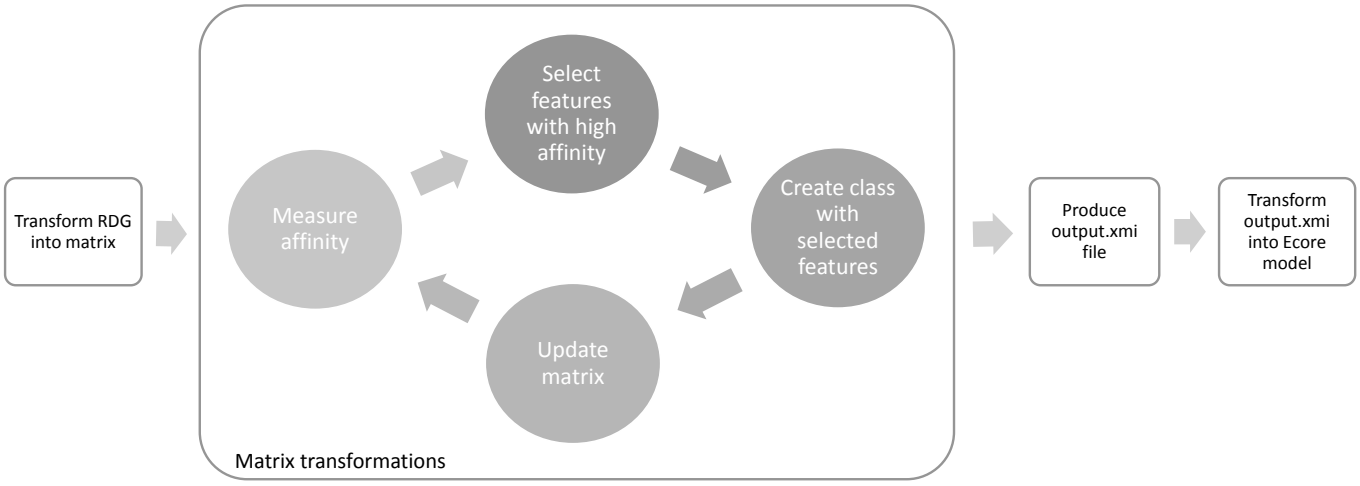


Figure 2: Transformation process

[3]Taken directly from the full case description, which can be found at https://github.com/martin-fleck/cra-ttc2016/blob/master/case_description/TTC2016_CRA.pdf

3.1 Initial Processing

The first step in the process is importing the Responsibility Diagram and parsing it in order to harvest the available features and existing relationships between them.

A matrix of size $\rho \times (\rho + \sigma)$ is generated, where ρ is the number of methods and σ is the number of attributes. The matrix will be filled with 0, 1 or 2 which represents the type of relationship between two particular features (no relationship, unidirectional and bi-directional respectively). This value will be referred to as α. The result after this step for Input A is shown in Table 1. In the example, relationships for each method with itself will be 1 but this can vary depending on certain properties (this concept is explained further under section 4).

Table 1 Associate Matrix for Input A - First Step

	Methods				Attributes				
	0	1	2	3	4	5	6	7	8
0	1	0	1	0	0	0	0	1	1
1	1	1	0	2	0	0	1	1	0
2	0	0	1	1	1	0	0	0	0
3	1	2	0	1	0	1	0	1	0

The distinction between uni- and bi-directional relationships in the associate matrix is the first difference from the classic Markov Clustering Algorithm implementation, which contemplates bi-directional links only.

Table 2 Associate Matrix for Input A - Second Step

	Methods				Attributes				
	0	1	2	3	4	5	6	7	8
0	1,4	0,2	1,1	0,2	0	0	0	1	1
1	1,2	1,5	0,1	2,4	0	0	1	1	0
2	0,1	0,1	1,3	1,1	1	0	0	0	0
3	1,2	2,4	0,1	1,5	0	1	0	1	0

In the next step, the algorithm analyzes the similarities between each pair of methods by counting how many features they share relationships with (both values should be > 0 to be counted). In the example, methods 1 and 3 have four features in common (0, 1, 3 and 7). Each value will be calculated for every pair of methods and placed next to each cell value separated by a comma, and will be referred to as β. This newly calculated value is arguably one of the most important across the entire algorithm, as it directly related to the stregth of the relationship between two given methods. As a result, values in each cell will correspond to α,β as shown in Table 2. Attribute values will remain unchanged in this step.

Shortly after, a new value δ will be calculated on each of the recently modified cells. This calculation, as well as the use of the diagonal will vary depending on various properties associated with the initial Responsibility Dependency Graph. This is explained more in depth in section 4.

However, in this case the affinity index (δ) will be defined as $\delta = \beta$. Results for this example can be found in table 3.

3.2 Class Generation

In order to achieve the last step of the algorithm and finally get to the resulting class diagram, the following calculations will be performed:

1. Sum total for each method row, showing which method has the most/strongest relationships.
2. Maximum affinity index -calculated previously- for each method column, showing which pair has the strongest relationship.
3. Sum total for each attribute column, showing by how many methods each attribute is used.

Results after this step are shown in Table 3.

From here on, we start by choosing the first row with the highest row total. If the matrix exceeds a specified threshold in the number of methods, values in this row will be slightly increased in order to simulate the generation of a superclass or "leading" class, but in the example shown values will remain the same.

Table 3 Associate Matrix for Input A - Third Step

	Methods				Attributes					Sum
	0	1	2	3	4	5	6	7	8	
0	4	2	1	2	0	0	0	1	1	11
1	2	5	1	4	0	0	1	1	0	14
2	1	1	3	1	1	0	0	0	0	7
3	2	4	1	5	0	1	0	1	0	14
Max/Sum	4	5	3	5	1	1	1	3	1	

For each method column in the row selected before, we check if the affinity index is equal to the previously calculated maximum. If the values are

the same, we store that method in an array in order to group them together in the class generation step. In the example, the first row selected will be Method 1 (with 14 as row sum), and the only candidate method will be itself. This attempts to ensure the highest possible cohesion within all method-to-method relationships.

After that we move on to the grouping of attributes to be put within the same class. In order to do so, we calculate the sum total of each attribute column but only for the methods previously selected. If the sum is equal to or greater than 50 percent of the column total, that attribute will be selected since the majority of methods that use it are candidates for the newly generated class. If the sum is less than 50 percent that means the attribute will produce higher cohesion placed in a different class. For Method 1, attribute 6 will be grouped with it since $1 \geq 0.5 \times 1$, but attribute 7 will not since $1 < 0.5 \times 3$.

Methods and attributes selected in the last section will be placed in a newly created class. In order to prevent other classes from re-using them in the future, column values for each attribute and row/column values for each method will be set to zero. This process will go on generating classes by grouping methods and attributes until values on all cells are zero.

After wrapping up the last set of features within a class, the solution will transform the class diagram into a .xmi file with the correct formatting and store it within the same file-path as the input.

4 The Diagonal Dilemma, Affinity Index Calculation and Leading Class

When analyzing and processing affinity indexes, values shown in the diagonal are often (such as in the example shown in this paper) necessary in order to generate a class diagram that carries a high CRA-index. However, in some cases it has been proven to be misleading and therefore largely increasing the overall coupling of the output.

For this reason, it has been determined the need to establish a threshold within the number of methods required in order to deem the diagonal counter-productive. For our solution, every time the input exceeds 12 methods the values in the diagonal will become zero. This threshold is not completely precise and might be refined after further experimentation.

Furthermore, for inputs in which the number of methods is < 12 the diagonal might require to be turned to zero as well for very specific cases (such as having less attributes than methods and thus requiring multiple methods to be put together in order to increase the cohesion). This avoids methods to be isolated and being grouped up only with the attributes they make use of.

When it comes to calculating the Affinity Index δ, several different heuristics using both α and β have been tested (such as $\alpha + \beta, \alpha^2 + \beta$, etc.) but the best results have been obtained by using simply $\delta = \beta$.

Another concept introduced in this algorithm is the generation of a main or leading class which contains the majority of the methods/attributes, while keeping the overall strongest relationships between features in the associate matrix to classes other than the main one. This is a means to represent the way in which developers generally build class diagrams when illustrating real-life schemes, having one class that contains a large number of methods and attributes (i.e. class Person/Student/Employee, class Product/Furniture/Vehicle, etc.) as well as a few other classes that provide functionality, usability or additional relevant information.

5 CRA Index and Execution Time

The results obtained by the proposed solution can be found below:

	Proposed Excel Solution[4]		TTC MOMoT Solution	
Input	CRA-Index	Execution Time	CRA-Index	Execution Time
A	3.0	0,46 sec.	3.0	4 min. 03 sec.
B	2.99999999	1,13 sec.	1.125	5 min. 05 sec.
C	0.64803921	1,92 sec.	-5.63571428	12 min. 02 sec.
D	-0.2913547	7,54 sec.	-23.6338095	26 min. 51 sec.
E	0.21916410	21,83 sec.	-69.65545274	38 min. 08 sec.
F	1.92302499	2 min. 46 sec.	Unknown	Unknown
G	-0.0343154	37 min. 42 sec.	Unknown	Unknown

Table 5 Class Diagram Generation Results for Proposed Solution compared to TTC's

[4]Executions have been performed on a Lenovo Y50 notebook running Windows 8 x64, i5 4210H @ 2.90GHz, 8.00GB

6 Visualization

The output .xmi file is finally transformed into an ecore model which can be later used for visualization through "Initialize ecore_diagram diagram file" feature in Eclipse. Figure 3 shows the class diagram for the output produced for Input A.

The proposed solution can be executed in a remote server through a Powershell prompt, allowing users to provide an input file and transform it into an output .xmi file or an ecore model.

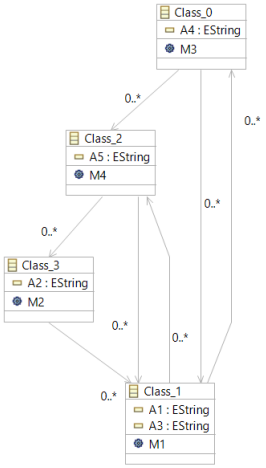


Figure 3: Class Diagram for Input A

7 Related work

There have been several attempts done by researchers to solve the CRA problem. Recently, meta-heuristic optimization techniques such as Genetic Algorithm (GA), Hill Climbing (HC), Simulated Annealing (SA), Particle Swarm Optimization (PSO) have been analyzed by many research groups [E07]. Multi-objective genetic algorithm (MOGA) is another technique that has been used to solve the Class Responsibility Assignment Case [BBL10] [MJ14].

8 Conclusion

Although the Markov Clustering Algorithm has been proven to be exceptionally useful for finding strongly connected components within bi-directional, un-weighted graphs, a classic implementation of this algorithm is not enough to completely cover the problem analyzed in this paper. Originally this algorithm merely compares similarities between features (which belong to an unique type), and expands the variables until they're stable enough to point out the clusters found in the graph. In-depth explanation of this algorithm can be found in its very own author's paper [SVD00].

Without the adjustments proposed in this paper, but specifically those described under section 4, Excel would simply return poorly generated class diagrams and the resulting CRA-index would be considerably lower.

When it comes to resulting CRA values, they're closely comparable to those obtained by other solutions in the competition, being located under highly accurate transformation tools such as VIATRA or Henshin. However these solutions are based on space-exploration strategies and genetic algorithms respectively, and therefore their execution time is significantly higher as they require to wander through several solutions and compare them afterwards.

Even though the results obtained are somewhat respectable, there still exists a gap between the class diagrams generated and the overall optimal class diagram for each input, which remains yet undisclosed. This gap could be filled by refining the proposed techniques, establishing new thresholds or simply adding new variables that could help achieve higher CRA-indexes.

References

[SVD00] S. M. van Dongen. Graph Clustering by Flow Simulation. 2000.

[KM09] K. Macropol. Clustering on Graphs: The Markov Clustering Algorithm. 2009.

[E07] A. P. Engelbrecht. Computational Intelligence: An Introduction, 2nd ed. John Willey & Sons. 2007

[BBL10] M. Bowman, L.C. Briand, Y. Labiche. Solving the Class Responsibility Assignment Problem in Object-Oriented Analysis with Multi-Objective Genetic Algorithms in IEEE Transactions on Software Engineering, vol. 36, no. 6, pp. 817-837, 2010

[MJ14] H. Masoud, S. Jalili. A clustering-based model for class responsibility assignment problem in object-oriented analysis in Journal of Systems and Software, volume 93, pp. 110-131, 2014

The SDMLib solution to the Class Responsibility Assignment Case for TTC2016

Christoph Eickhoff, Lennert Raesch, Albert Zündorf
Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73, 34121 Kassel, Germany

raesch|christoph|zuendorf@uni-kassel.de

This paper describes the SDMLib solution to the Class Responsibility Assignment Case for TTC2016. SDMLib provides reachability graph computation ala Groove. Thus, the simple idea was to provide rules for possible clustering operations and then use the reachability graph computation to generate all possible clusterings. Then, we apply the CRAIndex computation to each generated clustering and identify the best clustering. Of course, this runs into scalability problems, very soon. Thus, we extended our reachability graph computation to do an A* based search space exploration. Therefore, we passed the CRAIndex computation as a metric to our reachability graph computation and in each step, we consider the set of not yet expanded graphs and choose the one, that has the best metric value for expansion. The paper reports about the results we achieved with this approach.

1 Introduction

This paper describes the SDMLib solution to the Class Responsibility Assignment Case for TTC2016 [1]. SDMLib provides *reachability graph computation* ala Groove [2]. For a given start graph and a given set of rules, the reachability graph computation generates all graphs that may be derived from the start graph by applying all rules at all possible matches as often as possible in all possible orders. Each time a new graph is computed, we search through the set of already computed graphs for an already known isomorphic graph. As proposed by [2], SDMLib computes node and graph certificates which are then used as hash keys to access potentially isomorphic graphs efficiently. The node certificates then also help to do the actual isomorphism test. If a new graph has been generated, we create a so-called *reachable state node* and we connect the reachable state node of the predecessor graph with the reachable state node for the new graph via a rule application edge labeled with the name of the rule used. In addition, a root node of the graph is attached to the reachable state node. Altogether, the generated reachability graph has a top layer consisting of reachable state nodes connected via rule application edges and each reachable state node refers to the corresponding application graph via a `graphRoot` link. In SDMLib, this whole structure is again a graph, and graph rules may be applied to it in order to find e.g. reachable states with a maximal metric value for the attached application graph or to find states where all successor states have lower metric values or to find the shortest path leading to the best state. Actually, any graph related algorithm may be deployed.

The Class Responsibility Assignment Case challenges the rule orchestration mechanisms provided by the different model transformation approaches. Thus, our solution uses the SDMLib reachability graph computation for rule orchestration. This is a very simple way to apply all rules in all possible ways and in addition we are able to investigate all intermediate results in order to identify which paths through the search space are the most interesting ones. The drawback of this approach is that it wastes a lot of runtime and memory space for copying the whole class model graph each time a rule is applied and for the search of already known isomorphic copies of the generated graphs. As shown in the case description, the number

Submitted to:
TTC 2016

of possible clusterings grows with the Bell number, i.e. for larger examples a complete enumeration of all possible clustering is not possible in a meaningful time. As only a small fraction of the search space can be explored, it might be helpful to be able to investigate all intermediate states to identify the most promising spots for further expansion. Thus, we hope that the flexibility provided by the SDMLib reachability graphs to investigate different intermediate states pays off, in the end.

As it is usually not possible to generate the whole reachability graph for a given example, our reachability graph computation may be restricted to a maximum number of reachable states to be generated. Next, we have extended our reachability graph computation with an A* like search space exploration that takes a metric as parameter and at each step chooses the state with the best metric value for expansion. We have developed two variants of this A* algorithm which will be discussed below.

The next section introduces the rules we use to solve the Class Responsibility Assignment Case and then Section 3 shows the different search strategies we use in this example. Finally, Section 4 shows our performance measurements.

2 The Model Transformation Rules

Our feature clustering approach uses two SDMLib model transformation rules. In the preparation phase we use the rule shown in Figure 1 to create one class for each feature in our class model.

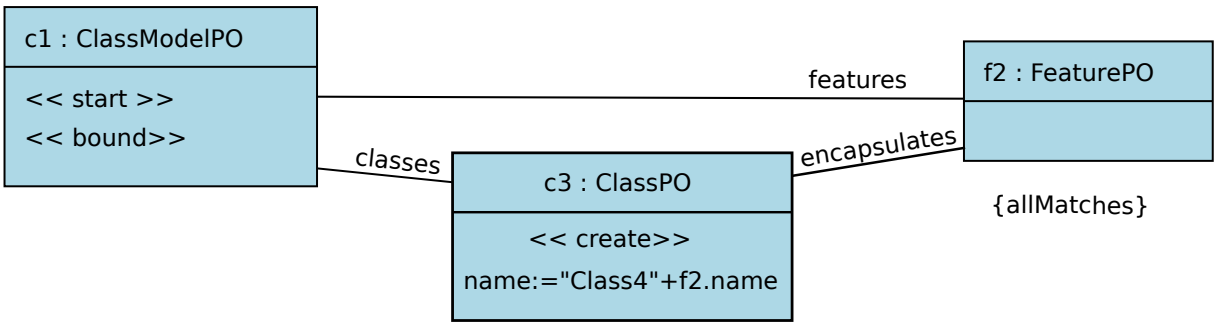


Figure 1: Rule adding initial classes

This rule starts by matching the pattern object `c1` to the `ClassModel` object passed as parameter. Then, `f2` is matched to a Feature object attached to this `ClassModel` object. The {`allMatches`} constraint causes the rule to be applied to all possible matches. Thus, for each `Feature` object in our current `ClassModel`, the <<`create`>> sterotype on pattern object `c3` causes the creation of a new `Class` object. In addition the new `Class` object is attached to the `ClassModel` via a `classes` link and to the `Feature` object via an `encapsulates` link. Finally, the new `Class` object's `name` attribute gets assigned the concatenation of the prefix `"Class4"` and the `name` of the current `Feature`. Thus, after the execution of this rule, each feature has its own class containing just this feature. This class model is then used as starting point for the repetitive application of our clustering rule.

Our clustering rule merges classes along functional or data dependencies, cf. Figure 2. The matching of this rule starts with the `ClassModel` object which is bound to `c1` at rule invocation. Then we follow a `classes` edge to find a match for `c2`, i.e. a `Class` object in our `ClassModel`. Next, we follow an `encapsulates` edge to match a `Method` object `m4` contained in `c2`. The object matched by `m4` must have a `dataDependency` edge or a `functionalDependency` edge to a `Feature` object matched by the pattern

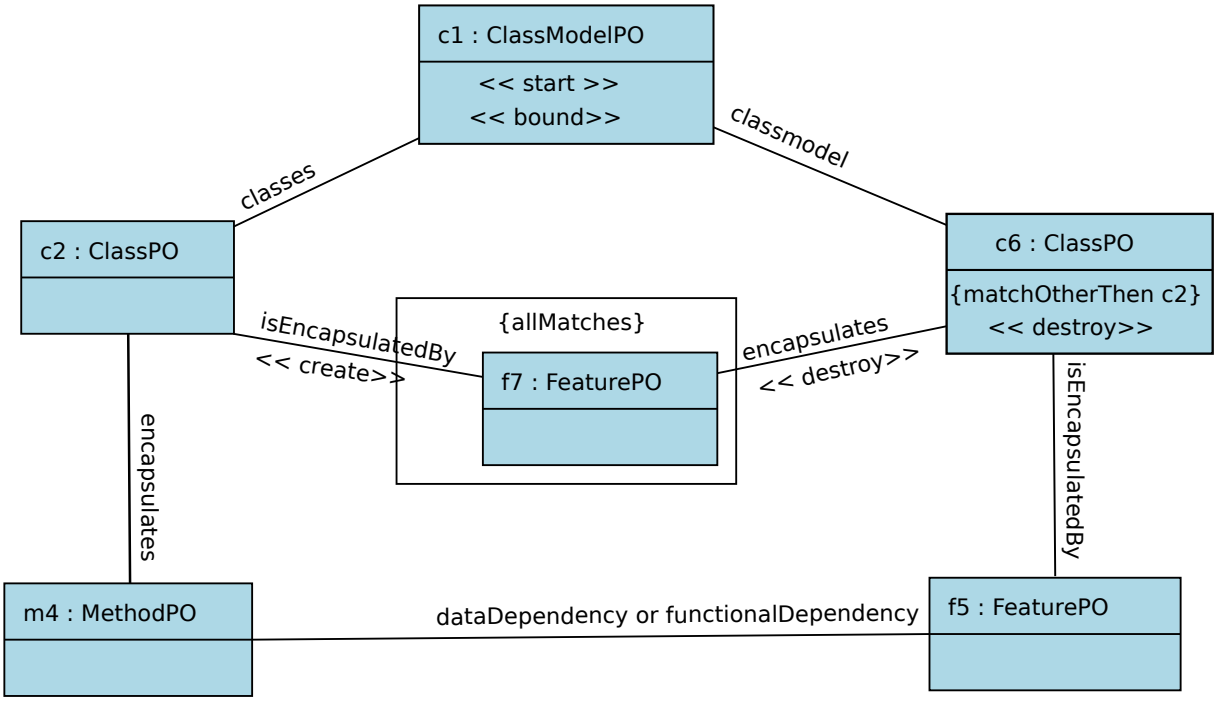


Figure 2: Merging Classes via Feature Dependencies

object `f5`. This `Feature` object in turn must be contained in a `Class` matched by `c6`. By default, SDMLib allows homomorphic matches, thus `c2` and `c6` would be allowed to match the same `Class` object. Via the {`matchOtherThen c2`} clause, we enforce isomorphic matching, i.e. `c2` and `c6` must match two different `Class` objects. Finally, the Class matched by `c6` must belong to our `ClassModel c1`. When such a match is found, the subpattern containing the `FeaturePO` pattern object `f7` is executed on all possible matches. Pattern object `f7` matches for all features contained in the `Class` matched by `c6`. (Note, `f7` exploits homomorphic matching and will also match the `Feature` object already matched by `f5`.) For each `Feature` object, the `encapsulates` edge connecting it to the `Class` matched by `c6` is deleted and a new `isEncapsulatedBy` edge connecting it to the `Class` matched by `c2` is created. After transferring all features to the `Class` matched by `c2`, the `Class` matched by `c6` is destroyed. Thus, the rule shown in Figure 2 merges two classes that are connected via a feature dependency into one class.

Our clustering rule merges classes only if there is a dependency between them. This already utilizes application specific knowledge about our CRAIndex metric. Our search space expansion will start with classes containing only one feature each. Merging classes without a dependency between them is not going to improve the CRAIndex of the resulting class model. Merging classes has the potential to improve the CRAIndex only if the classes contain features that depend on each other. Thus, our clustering rule is already optimized for the optimization of the CRAIndex. Using a different metric would perhaps require a more general clustering strategy. As the metric is evaluated during the search space exploration, it would be easy to simply merge any two classes, as any graphs resulting from applying a non metric improving rule would immediately be dismissed anyways.

3 The Search Space Exploration Mechanisms

At the beginning, a `ReachbilityGraph` object is initialized with a start graph or `startState` and with a set of `rules` that shall be applied to the different reachable states. Our standard A* search space exploration algorithm is shown in Listing 1. For the A* search space exploration, we call e.g.

```
rg.explore(25000, g -> CRAIndexCalculator.calculateCRAIndex((ClassModel) g));
```

where the first parameter is the maximal number of states to be generated and the second parameter is the metric function that guides our search space expansion algorithm. First, our expansion algorithm initializes its `todo` list with the `startState` and adds the `startState` to a hash table of reachable `states` where a graph `certificate` is used as key, as proposed by [2]. Then, line 5 loops through the `todo` list until it drains or the maximal `depth` of the search space is reached. Line 6 of our explore method just `sorts` the `todo` list before choosing a new element in line 7. Thereby, each exploration step considers the state with the best CRAIndex for further expansion resulting in a depth first like expansion strategy. Then line 8 and line 9 iterate through all `rules` and all matches. For each match, we `clone()` the `current` state and `apply()` the rule changes to that clone, resulting in a `newState`. As the `newState` may have been created by other rule applications already, line 11 tries to `find` an `isoOldState`, i.e. the `find` operation computes the `certificate` of the `newState` and tries to look it up in the `states` hash table. This involves an isomorphism check to exclude accidentally matching certificates. If no `isoOldState` is found, line 13 adds the `newState` to the hash table of reachable `states`, line 14 adds an edge labeled with the applied rule from the `current` state to the `newState`, and line 15 adds the `newState` to our `todo` list. If there is an `isoOldState` line 17 just adds an edge from the `current` state to the `isoOldState`.

```
1  ReachabilityGraph::explore(depth, metric) {
2     todo = new ArrayList();
3     todo.add(this.startState);
4     states.put(certificate(this.startState), startState);
5     while (! todo.isEmpty() && states.size() <= depth) {
6        sort(todo, metric);
7        current = todo.get(0); todo.remove(0);
8        for(Rule r : this.rules) {
9           while (r.findMatch()) {
10             newState = current.clone().apply(r);
11             isoOldState = find(states, newState);
12             if (isoOldState == null){
13                states.put(certificate(newState), newState);
14                addEdge(current, r, newState);
15                todo.add(newState);
16             } else {
17                addEdge(current, r, isoOldState);
18             }
19          }
20       }
21    }
22 }
```

Listing 1: Default A* based Search Space Expansion

Within each step, our default A∗ search space exploration strategy generates all successors of the `current` state. For case E there are about 400 dependencies, thus, the initial state has about 400 successor states. While this number decreases by one with each rule application, the first 100 rule applications have 350 successors on average resulting in 35000 states, which already exceeds our memory space. To improve this, we added a variant of our algorithm called *Ignore Decline* mode. The Ignore Decline mode improves our exploration algorithm by comparing the metric value of the `newState` with the current `bestMetric`. If the metric of the `newState` is lower then the `bestMetric` we ignore the `newState`, i.e. we do not add it to our reachability graph nor to our `todo` list. This may exclude some important candidates from later consideration but it reduces the number of states to be added to our reachability graph considerably thus reducing memory space consumption.

The second variant of our search space exploration algorithm is called *Promote Improvements* mode. The Promote Improvement mode computes the metric for each new state and in case of an improvement compared to the current state, we stop the expansion of the current state (putting it back into the todo list). Then we jump back to line 6 and start with a new iteration, i.e. we sort the todo list (bringing the new best state to the front) and continue the search space exploration with this new state. The Promote Improvement mode reaches local optima of the reachability graph very fast. Note, when the search is exhausted for some state and we go back to earlier states, rule application on those earlier states will first produce the same matches as in earlier runs. These same old matches will be identified by line 11 as `isoOldStates` and thus ignored. However, this requires the computation of a certificate and an isomorphism check. To avoid this effort, our real implementation of the Promote Improvement mode stores the number of already created successors for each state and on reconsideration, this number of rule applications is directly ignored.

4 Performance Results

Figure 3 shows the CRAIndex we achieve for the different input models and the time our best algorithm needed to compute this.

	Input A	Input B	Input C	Input D	Input E
CRA Index	3	3,75	2,94	0,49	0,43
Performance	00:00.412	00:00.665	04:14.479	32:11.173	04:56.309

Figure 3: Performance Summary

References

[1] M. Fleck, J. Troya, and M. Wimmer. TTC2016 The Class Responsibility Assignment Case. https://github.com/martin-fleck/cra-ttc2016, 2016.

[2] A. Rensink. The GROOVE simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance*, pages 479–485. Springer, 2003.

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

1 Introduction

In this paper we describe a solution for the Class Responsibility Assignment(CRA) case [3] of the 2016 Transformation Tool Contest using the MDEOptimiser[1] [4] tool prototype. 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 in Section 2. Section 3, then, presents our solution to the challenge case, which we evaluate in Section 4 based on the criteria defined with the case. Section 5 discusses lessons learned from this experiment and highlights future research.

2 Case Study Description

In this section, we give a brief overview of the CRA case.

The challenge is about a key step in object-oriented software 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 a set of classes with unique names and an allocation of features to these classes that minimises dependencies between classes.

3 Solution Overview

We separate the problem into two sequentially solved sub-problems: find an optimal allocation of features to a suitable set of classes and ensure all classes have unique names.

The first problem is search based, and is solved with MDEOptimiser. The second problem is a simple transformation problem, which we solve using a simple iteration expressed in Xtend. In the following two subsections, we discuss each solution in turn. The complete solution is also available on SHARE [1]. The complete experiment dataset is also available.[2]

3.1 Solving the search problem

Any search problem is composed of 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.

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

[1] `https://github.com/mde-optimiser/mde_optimiser`

[2] DOI: 10.18742/RDM01-47

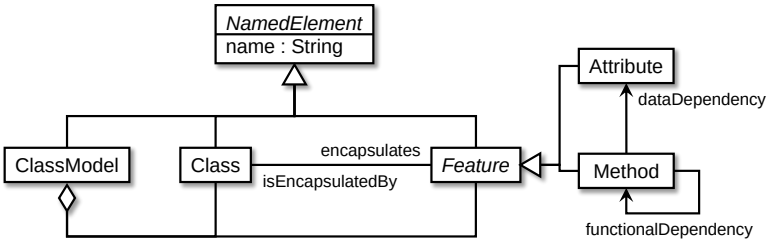


Figure 1: Metamodel describing the search space of the class–responsibility assignment problem

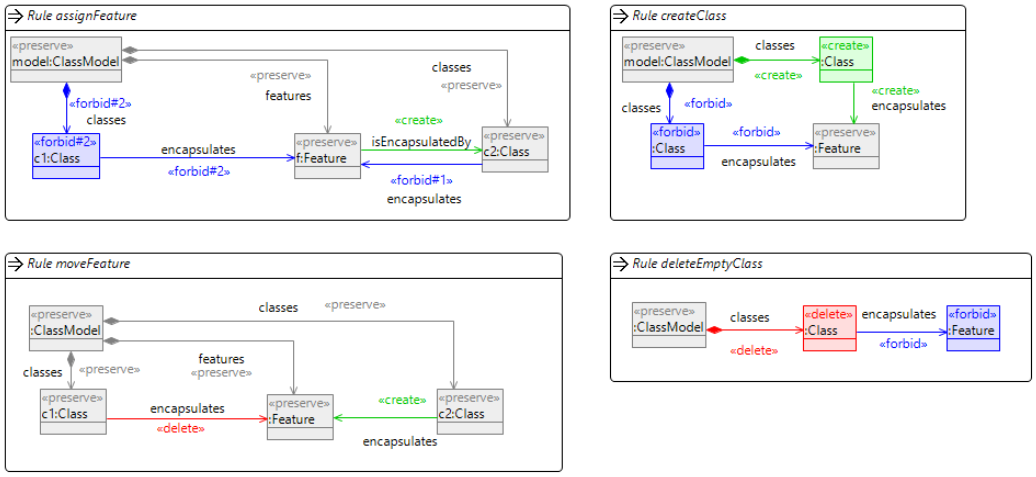


Figure 2: Model evolvers used for the challenge case

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

3.1.2 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 solution 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. 3.2.

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

```
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 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.

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.

3.1.3 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 3 objective functions: minimise number of unencapsulated features; minimise number of empty classes; maximise CRA (a combination of cohesion and coupling metrics as defined in the challenge case).

3.1.4 Case solution

Figure 3 shows how the CRA problem is specified in MDEOptimiser. In Line 3, we define the search space through the metamodel. 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 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

Table 1: Product-criteria measurements

Model	Configuration	Avg. time	Best CRA	Fleck
A	I	1m 28s 447.5ms	1.666666667	1.75
	II	4m 46s 712.7ms	1.666666667	
B	I	3m 25s 905.5ms	1.205555556	3.083
	II	9m 05s 95.2ms	1.205555556	
C	I	5m 53s 336.5ms	0.3952380952	-3.79
	II	35m 16s 177.6ms	0.01060606061	
D	I	14m 58s 558ms	-43.829167	-23.63381
	II	2h 40m 09s 57.5ms	1.599794239	
E	I	14m 28s 123.1ms	N/A	-66.65545
	II	3h 07m 47s 617.1ms	N/A	

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.

3.2 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.

4 Evaluation

In this section, we evaluate our solution against the criteria defined in the challenge case.

4.1 Product criteria

Here, we report the results 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 10 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 two 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[3].

2. *No support for breeding:* Breeding (i.e., creating a new candidate solution by intermixing aspects of two good parent solutions) is not yet supported in MDEOptimiser, making it more difficult for the search algorithm to escape from local maxima. Hence, large parts of the search space may never be reached.

Table 1 shows an overview of the results 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[4], so we show their CRA values for comparison.

For the less complex models, we could already obtain reasonable results with a smaller number of generations and a smaller population size. For model C, our CRA in Configuration I is 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 except for model D, for which we obtained a better CRA with Configuration II, but at a substantial time cost. Indeed, it appears from the table that we get worse results for Configuration II for model C and the same CRAs for models A and B. 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 weeded out earlier. For model E, we were unable to find any valid models using any of the configurations. In an earlier run (with a

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

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

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.

4.2 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.

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 **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.

5 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. 4.1 a number of things we will investigate to avoid this.
- *Additional features required.* As a very new tool, MDEOptimiser lacks a number of important features. For example, we currently do not support Henshin rule parameters, which would have enabled us 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, we had to separate the overall transformation into two phases, which could be argued to be a better design. More generally, we need to identify 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.

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] Javier Troya Martin Fleck and Manuel Wimmer. The class responsability assignment case. In *9th Transformation Tool Contest (TTC 2016)*, 2016.

[4] Steffen Zschaler and Lawrence Mandow. Towards model-based optimisation: Using domain knowledge explicitly. In *Proc. Workshop on Model-Driven Engineering, Logic and Optimization (MELO'16)*, 2016.

Class Responsiblity Assignment Case: a VIATRA-DSE Solution*

András Szabolcs Nagy
nagya@mit.bme.hu

Gábor Szárnyas
szarnyas@mit.bme.hu

Budapest University of Technology and Economics
Department of Measurement and Information Systems
MTA-BME Lendület Research Group on Cyber-Physical Systems

Abstract

This paper presents a solution for the Class Responsibility Assignment Case of the 2016 Transformation Tool Contest. The task is to assign features (methods and attributes with dependencies to each other) to classes and optimize a software metric called CRA-Index. The solution utilizes the rule-based design space exploration framework VIATRA-DSE with the Non-dominated Sorting Genetic Algorithm (NSGA-II) and it extends the framework with a domain-specific state encoder to identify similar solutions and to obtain better performance. Furthermore, it also uses a domain-specific mutation operator and a slightly modified version of the provided transformation rule.

1 Introduction

Automated model transformations are a key factor in modern model-driven system engineering. Model transformations allow the users to query, derive and manipulate large industrial models, including models based on existing systems, e.g., source code models created with reverse engineering techniques. Since such transformations are frequently integrated with modeling environments, they need to feature both high performance and a concise programming interface to support software engineers.

Design space exploration (DSE) aims to explore different design candidates with respect to well-formedness constraints and objectives to aid system engineers in finding the best design or to dynamically reconfigure a system at runtime. While DSE has a long history (20–30 years) [8], especially for embedded systems, it has been adapted to model-driven system engineering only in recent years (discussed as related work in [1]).

VIATRA [2, 5] aims to provide the tooling support needed for these challenges by 1) an expressive model query language, 2) a carefully designed API for transformations and 3) a design space exploration tool easily integrated to the model-driven design process.

This paper presents a solution using VIATRA for the TTC 2016 Class Responsibility Assignment Case [7] (see Section A.1) which can be formalized as a DSE problem. The source code of the solution is available as an open-source project.[1] Additionally, there is a SHARE image available with the source code and scripts to run the solution on the provided input models.[2]

*This work was partially supported by the MTA-BME Lendület Research Group on Cyber-Physical Systems.

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

[1] `https://github.com/FTSRG/ttc16-cra-viatra-dse`

[2] `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=ArchLinux64_TTC-Arch-CRA-VIATRA.vdi`

2 Background of the Solution

VIATRA-DSE is a rule-based DSE framework [1, 9], which can explore different design candidates satisfying multiple criteria with respect to multiple objectives using graph transformation rules.

2.1 Approach of Rule-Based DSE

Figure 1 gives an overviews the most important concepts of this approach. A rule-based DSE problem consists of the following parameters:

- an *initial model* M_0,
- a set of *transformation rules* R that defines how the initial model (and the current model) can be manipulated,
- a set of *well-formedness constraints* C and
- a set of *objectives* O to optimize (minimize or maximize).

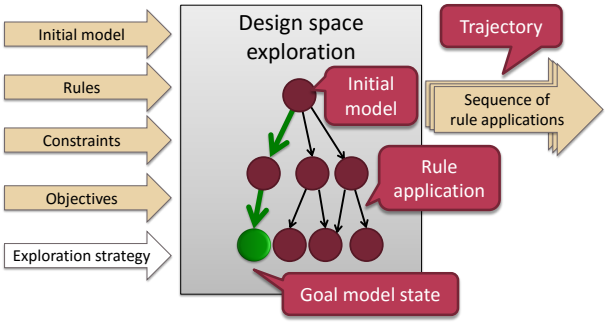


Figure 1: Overview of the rule-based DSE approach.

A solution for such a problem is a sequence of transformation rule applications (also called *trajectory*), which transforms the initial model M_0 to a model M_s, which satisfies all the well-formedness constraints in C. This solution is expected to be optimal (high-quality) with respect to objectives O.

The key strengths of this approach are that 1) models are attributed typed graphs and rules are graph transformation rules, which allows tight integration with model-driven system design, 2) the solution (i.e., the trajectory) also describes how to reach the found model from the initial state, which is important in certain problems, (e.g., runtime reconfiguration of a system needs to answer how to reach the candidate configuration) and 3) objectives can be derived from the model directly using even black box tools and from the trajectory as well.

To solve such a problem, a solver has to traverse a state space (also called design space) with an *exploration strategy*. This state space has an initial state representing the initial model and further states can be reached by applying the transformation rules. The state space can be infinitely large (e.g., a rule can make elements without an upper bound) and it can contain cycles (e.g., a rule can delete elements that an other rule just created). In Appendix A, Figure 4 shows a partial design space of a small CRA problem, where there are three methods and a single attribute.

2.2 VIATRA

VIATRA is an open-source Eclipse project written in Java and Xtend [6] and it builds upon the Eclipse Modeling Framework [4]. The VIATRA project provides the following main features:

- A declarative language for writing queries over models, which are evaluated incrementally upon model changes (formerly known as EMF-INCQUERY).
- An internal domain-specific language over the Xtend [6] language to specify both batch and event-driven, reactive transformations.
- A complex event-processing engine over EMF models to specify reactions upon detecting complex sequences of events.
- A rule-based design space exploration framework to explore design candidates with transformation rules where the design candidates must satisfy multiple criteria (presented in Section 2.3).
- A model obfuscator to remove sensitive information from a confidential model, e.g., for creating bug reports.

2.3 VIATRA-DSE

VIATRA-DSE provides an easy way to specify a rule-based DSE problem and to extend it with domain-specific needs. The condition (left hand side) of the transformation rules can be specified by the VIATRA Query language and the operation (right hand side) by simple Java code. Both constraints and objectives can be specified either by the VIATRA Query language or by any custom Java code. Furthermore, it supports the calculation of objectives both on the actual model and on the trajectory as well (e.g., executing certain rules has a cost).

VIATRA-DSE has several built-in strategies such as depth-first search, breadth-first search for systematic full exploration of the design space, fixed-priority search which uses priorities assigned to rules, and has metaheuristic strategies such as hill climbing and evolutionary algorithms, including Non-dominated Sorting Genetic Algorithm (NSGA-II) [3] and Pareto Envelope-based Selection Algorithm (PESA) [10]. Custom, domain-specific strategies can be integrated as well.

To recognize similar model states it uses a state encoding technique, which encodes model states into a textual representation. While these state codes can be easily compared to each other, the efficiency of the encoding process has a great impact on the exploration time. Additionally, rule applications are also encoded into a textual representation called activation codes. This allows to store a trajectory by its activation codes and to re-execute it later on an arbitrary model. VIATRA-DSE has a built-in generic state coder, which works out-of-the-box for most use cases, but it can be exchanged with a custom, domain-specific state coder, which can improve the performance of the exploration.

The framework is also capable of parallel exploration – at the time of writing the depth-first search, breadth-first search and the evolutionary exploration algorithms exploit this feature.

3 Implementation

In this section, we provide a detailed description of how we instantiated the task as a rule-based DSE problem.

3.1 Transformation Rules

Our solution uses the two transformation rules provided by the case, namely the *createClass* and *assignFeature* rules, however we enhanced the *createClass* rule with the following two modifications:

1. A class can be created only if there are no empty classes in the current model. This allows to prune the search space without losing any solutions.
2. Newly created classes are given a name *CX*, where *X* is a number depending on how many classes were created on the actual trajectory. This ensures that the classes have a unique name.

3.2 Well-Formedness Constraints

While the modified *createClass* rule ensures the unique name of the classes, we used VIATRA Query to capture the other two constraints (see Section A.3).

3.3 Objectives

We use two objective functions: one that calculates the CRA-Index of a class diagram and one that measures the violations of well-formedness constraints. The CRA-Index is calculated in the provided way, except we use VIATRA Query to calculate *MAI* and *MMI* incrementally upon model change. The other fitness function measures the number of unassigned features and it shall be minimized. This helps the exploration to reach a solution more easily.

These two objectives create a multi-objective optimization problem, which makes comparing solutions non-trivial. In this solution, we use the domination function [3] to compare solution candidates: a solution candidate s_1 dominates an other solution candidate s_2 if there is an objective function o_i that $o_i(s_1) > o_i(s_2)$ and for any other objective $o_j \neq o_i : o_j(s_1) \geq o_j(s_2)$. This approach will find a well-formed solution and an ill-formed solution candidate equal (in the sense of quality) if the ill-formed solution candidate has a higher CRA-Index.

3.4 Exploration with NSGA-II

For the exploration strategy, we used the NSGA-II genetic algorithm [3] crafted for multi-objective optimization problems. This algorithm maintains a population, i.e., a set of solution candidates (trajectories), and modifies them with genetic operators (mutations and crossovers) to derive new solution candidates. In an iteration, it combines the previous population and a newly created population and selects the best solution candidates to produce the next generation. The adaption of this algorithm as a rule-based DSE strategy can be found in [1].

We configured the NSGA-II strategy in the following way:

- The first population is generated by a breadth-first search algorithm selecting a trajectory into the population with a given probability and with a minimal length of 2. A population consists of 40 individuals.
- The following genetic operators are used, where mutations are used 80% of the time:

1. A mutation that adds a random rule application to the end of a trajectory.
2. A mutation that changes a random rule application in the trajectory to an other rule application.
3. Cut and splice crossover, which exchanges the tails of two trajectories creating two child trajectories.
4. Swap rule application crossover, which exchanges one activation code in the parent trajectories.
5. A *custom domain-specific mutation operator* that removes all *createClass* rule applications, where the created class remained unused. This helps the algorithm to find a well-formed solution.

- The stop condition consists of two sub-conditions that have to be fulfilled at the same time: 1) in the current population there is at least one solution that survived 100 iterations (i.e., the exploration cannot create a better solution) and 2) there is a well-formed solution in the current population.

As trajectories are encoded by a sequence of activation codes, for example a *swap rule application crossover* is executed by 1) randomly choosing the rule applications (activation codes) to swap, 2) creating the first child by executing the new trajectory, omitting rule applications that are infeasible, 3) backtracking to the initial model and 4) creating the second child as in step 2.

3.5 State Encoding

The solution uses a custom domain-specific state coder as the built-in state coder failed to recognize similar solutions. For example, if there are two methods $M1$ and $M2$ that are assigned to two different classes $C1$ and $C2$, then the built-in state coder creates a state code $C1(M1), C2(M2)$ or $C1(M2), C2(M1)$ depending on the trajectory, which are eventually representing the same solution (also the actual state code is much longer and redundant). Using this state coder, NSGA-II can store duplications in the population, which decreases efficiency. Thus, we created a domain-specific state coder that encodes model states leaving out the identifiers of the classes, encoding only the grouping of the features: $(M1), (M2)$. This state coder also has better performance reducing the exploration time.

3.6 An Alternative Solution

We also experimented with an other approach, where first we created a class for each feature in the initial model using the VIATRA Model Transformation API. Then we ran the exploration with a single rule that merges two classes. While, this approach could generate good solutions with positive CRA-Index with approximately in the same time, the presented solution produces better results.

4 Evaluation

4.1 Setup

As NSGA-II is a metaheuristic algorithm, it cannot provide a consistent solution for each run and runtime may vary because of the adaptive stop condition. Thus, we run the exploration 30 times for each input model and consider the median of the found fitness values and the median of runtime as result. This allows us to easily compare our results to other contestants' work.

The benchmarks were conducted on a 64-bit Windows 10 PC with a 2.50 GHz Intel i5-2450M CPU and 8 GB of RAM using Oracle JDK 8, 256 MB maximum heap size and four threads to exploit the four logical cores.

4.2 Results

Optimality: Figure 2 shows a box plot for the CRA-Index of the generated solution models for each input model. The smallest input model is solved pretty consistently, with a CRA-Index of 3. While for input model B, most of the runs returns a solution with a CRA-Index around 3.75, for the more complex input models the result varies greatly. However, the found solution models always have a positive CRA-Index disregarding a few outliers.

Performance: Figure 3 shows exploration times of the different runs in seconds on a logarithmic scale. The median values are marked with red. The runtime of the exploration varies greatly, especially on the largest input model E. While it could find a solution in about 4 minutes, in the worst case it needed one and a half hour. An interesting property of the input model C is that while it has twice as many features and thrice as many dependencies than input model B, the exploration time is just slightly longer.

Table 1 presents our aggregated results for each input model as stated in Section 4.1. We also included the metric of the best solutions our approach could find, to show that if there is no limit for execution time it can produce even better solutions.

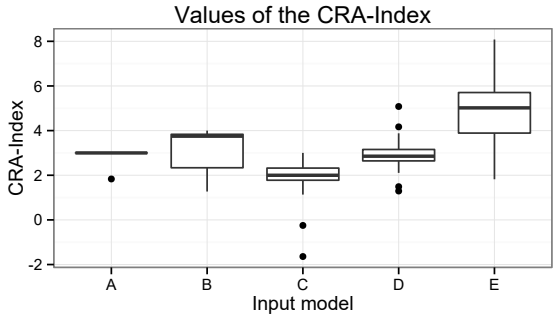


Figure 2: CRA-Index values by input model.

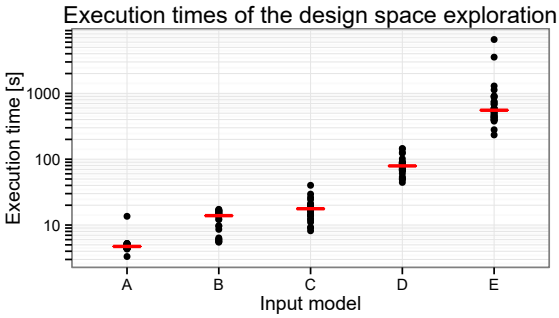


Figure 3: Execution times by input model with the median values marked with red.

	A	B	C	D	E
CRA-Index (best)	3	4	3.002	5.08	8.0811
CRA-Index (median)	3	3.75	1.9992	2.8531	5.0188
Time (median)	00:04.729	00:13.891	00:17.707	01:19.136	09:14.769

Table 1: Optimality and performance results of 30 runs.

5 Summary

This paper presented a complete solution for the Class Responsibility Assignment case of the 2016 Transformation Tool Contest. The approach of rule-based DSE and the VIATRA-DSE framework proved to be efficient for modeling the problem and sufficient for solving the case. The solution could be improved by gaining a deeper understanding of the CRA-Index metric and by adding a supplementary heuristic to the exploration.

References

[1] H. Abdeen, D. Varró, H. A. Sahraoui, A. S. Nagy, C. Debreceni, Á. Hegedüs, and Á. Horváth. Multi-objective optimization in rule-based design space exploration. In *ACM/IEEE Inter. Conf. on Autom. Soft. Eng.*, pages 289–300, 2014.

[2] G. Bergmann, I. Dávid, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró. VIATRA 3: A reactive model transformation platform. In *8th International Conference on Model Transformations*. Springer, 2015.

[3] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Computation*, 6(2):182–197, 2002.

[4] Eclipse.org. Eclipse Modelling Framework (EMF). `https://www.eclipse.org/emf/`.

[5] Eclipse.org. VIATRA Project. `https://www.eclipse.org/viatra/`.

[6] Eclipse.org. Xtend – Modernized Java. `https://www.eclipse.org/xtend/`.

[7] M. Fleck, J. Troya, and M. Wimmer. The class responsibility assignment case. In *9th Transformation Tool Contest (TTC 2016)*, 2016.

[8] M. Gries. Methods for evaluating and covering the design space during early design development. *Integration*, 38(2):131–183, 2004.

[9] Á. Hegedüs, Á. Horváth, and D. Varró. A model-driven framework for guided design space exploration. *Autom. Softw. Eng.*, 22(3):399–436, 2015.

[10] J. D. Knowles, R. A. Watson, and D. Corne. Reducing local optima in single-objective problems by multi-objectivization. In *Evolutionary Multi-Criterion Optimization, First International Conference*, pages 269–283, 2001.

A Appendix

A.1 Case Description in a Nutshell

The problem to solve is a simplified version of the class responsibility assignment (CRA) problem. As an input model, a set of attributes and methods are given with dependencies between them, in particular, methods can use certain attributes and other methods. The task is to assign all these features to classes with the goal of optimizing a software metric called *CRA-Index*. The CRA-Index is an objective function to maximize and it combines the cohesion ratio (inner dependencies of a class divided by the cardinality of the features) and coupling ratio (dependencies between two classes divided by the cardinality of the features) of the class diagram.

Contestants are given five models with increasing complexity to solve and they are to produce the corresponding high-quality models by using model transformation tools. The resulting models also have to satisfy the following constraints: 1) all features have to be assigned, 2) classes must have a unique name and 3) empty classes are not allowed.

The full description can be found in [7].

A.2 Design Space

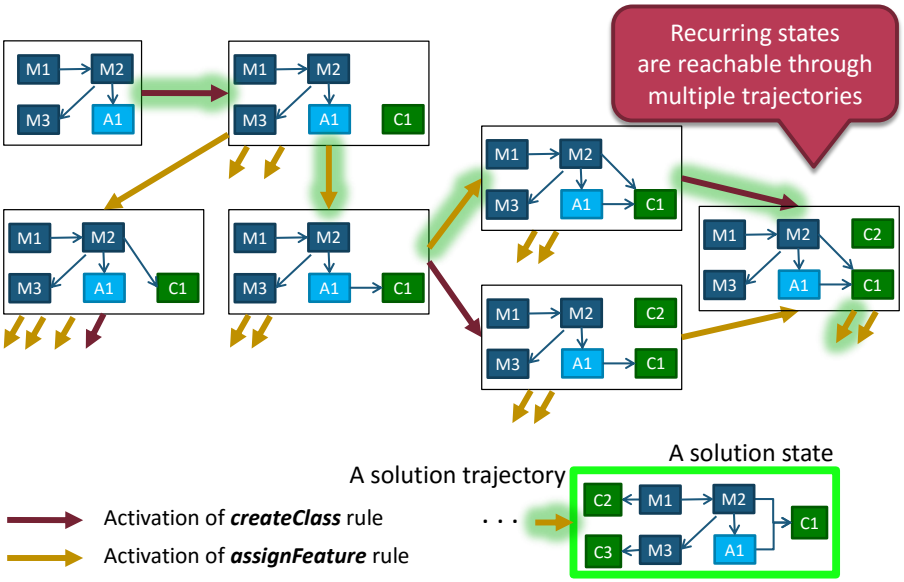


Figure 4: A part of the DSE state space and a solution trajectory.

A.3 Well-Formedness Constraints Captured by VIATRA Query

```
pattern allFeatureEncapsulated() {
    neg find notEncapsulatedFeature(_);
}

pattern noEmptyClass() {
    neg find emptyClass(_);
}
```

Well-formedness constraints

```
pattern notEncapsulatedFeature(f : Feature) {
    neg find encapsulated(_, f);
}

pattern emptyClass(c : Class) {
    neg find encapsulated(c, _);
}

pattern encapsulated(c : Class, f : Feature) {
    Class.encapsulates(c, f);
}
```

Helper patterns

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.

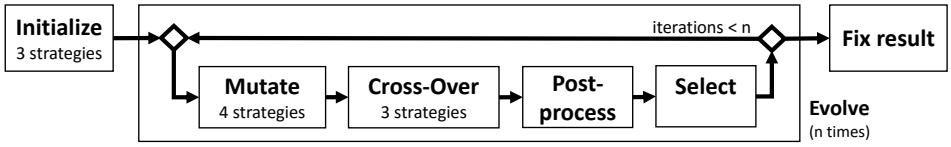


Figure 1: Overview

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

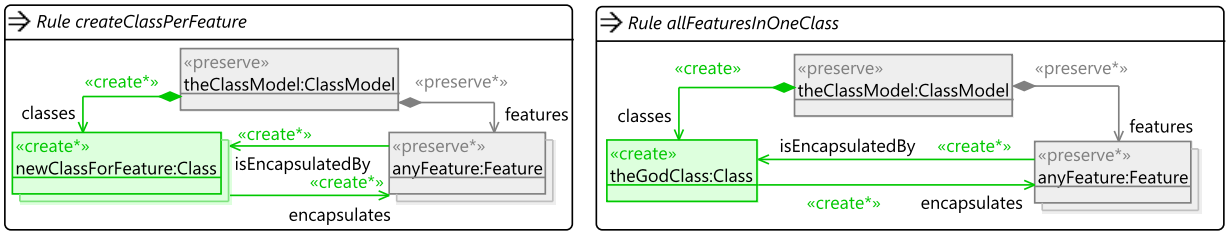


Figure 2: Rules to create the initial population

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.

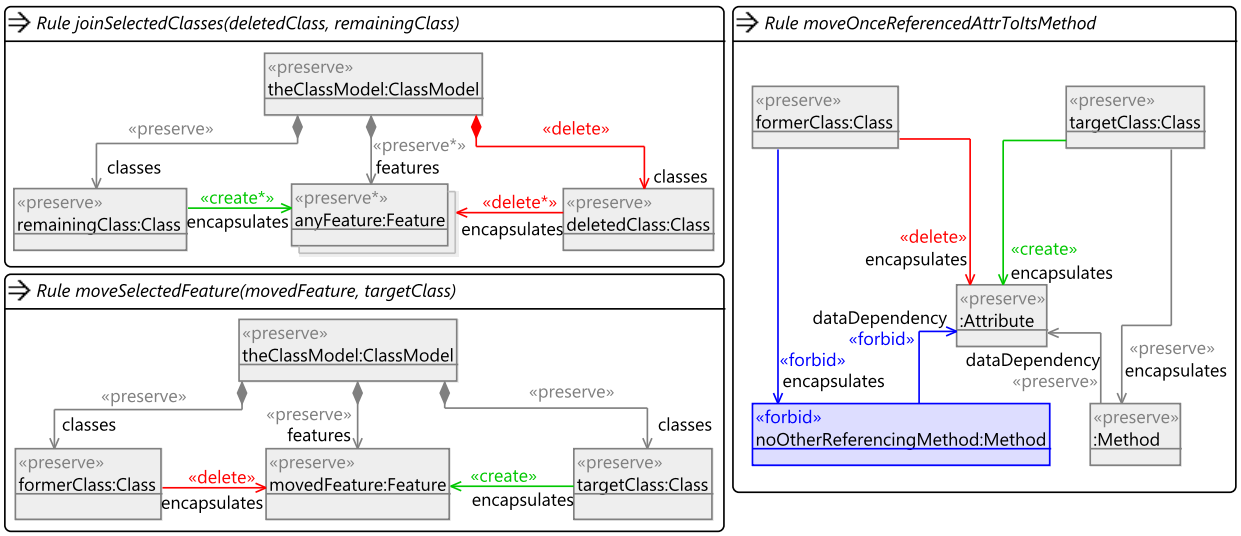


Figure 3: Rules for the mutation operator.

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

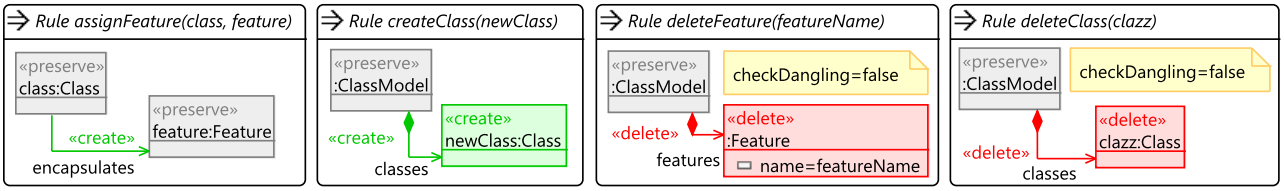


Figure 4: Utility rules for crossover (and mutation) operators.

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.

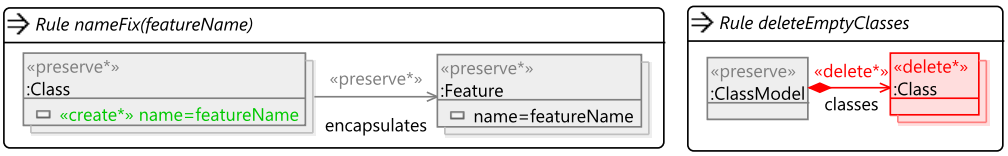


Figure 5: Rules for *fix result*.

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.

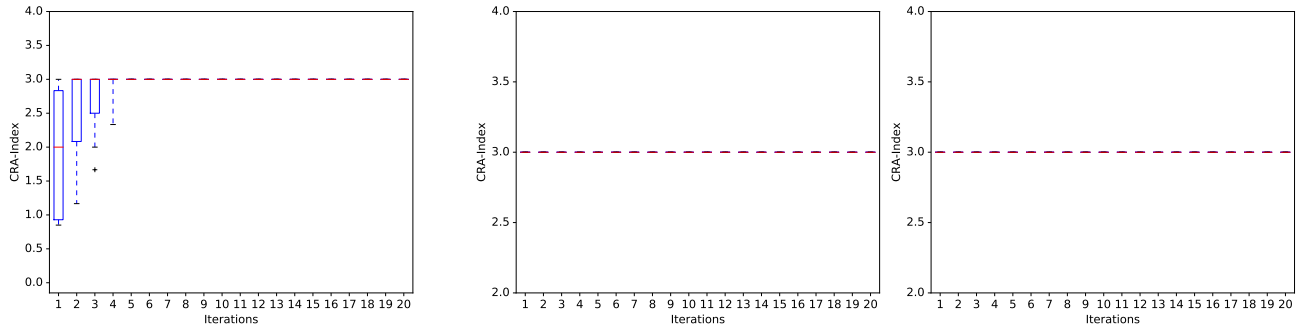


Figure 6: CRA of input model A depending on the selected initialization *allFeaturesInOneClass*, *oneClassPerFeature* or *mixed* (from left to right).

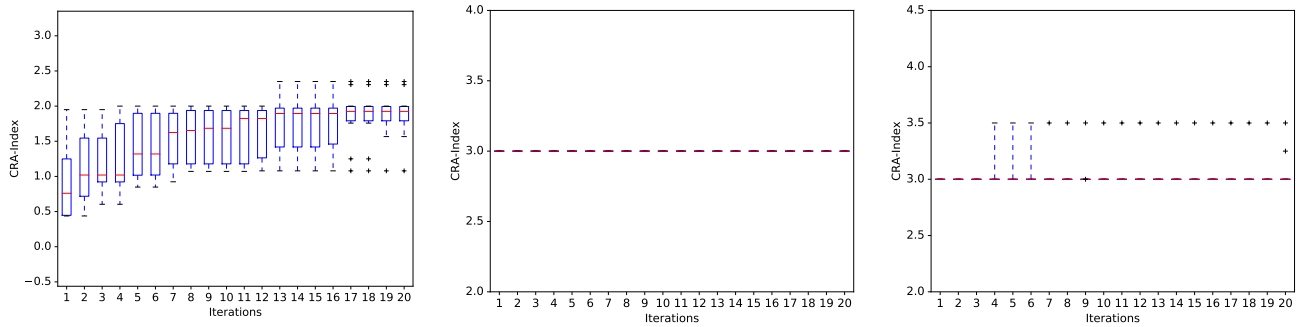


Figure 7: CRA of input model B depending on the selected initialization *allFeaturesInOneClass*, *oneClassPerFeature* or *mixed* (from left to right).

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

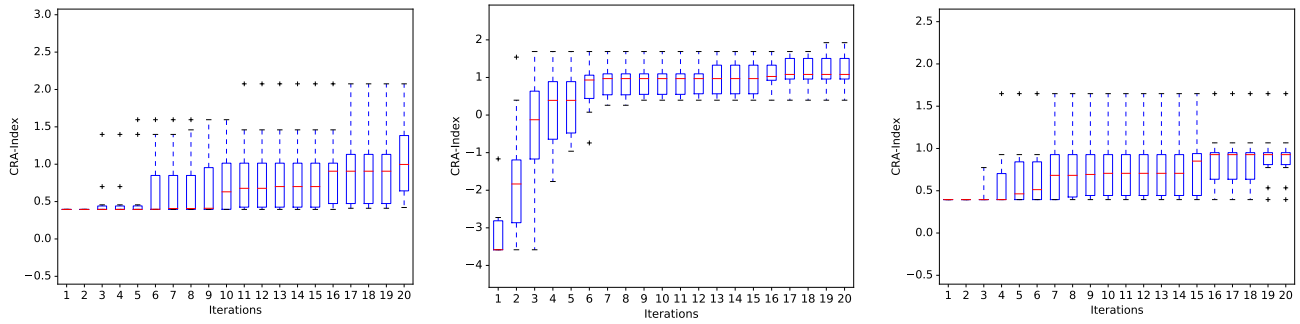


Figure 8: CRA of input model C depending on the selected initialization *allFeaturesInOneClass*, *oneClassPerFeature* or *mixed* (from left to left).

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.

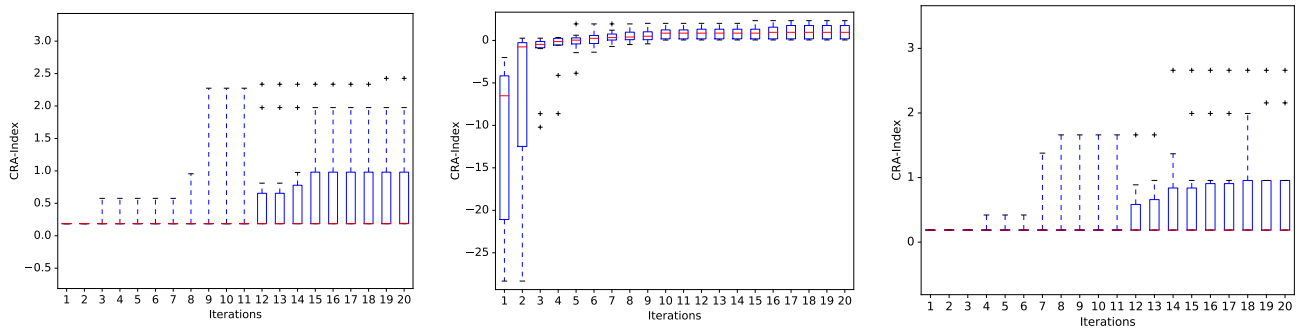


Figure 9: CRA of input model D depending on the selected initialization *allFeaturesInOneClass*, *oneClassPerFeature* or *mixed* (from left to left).

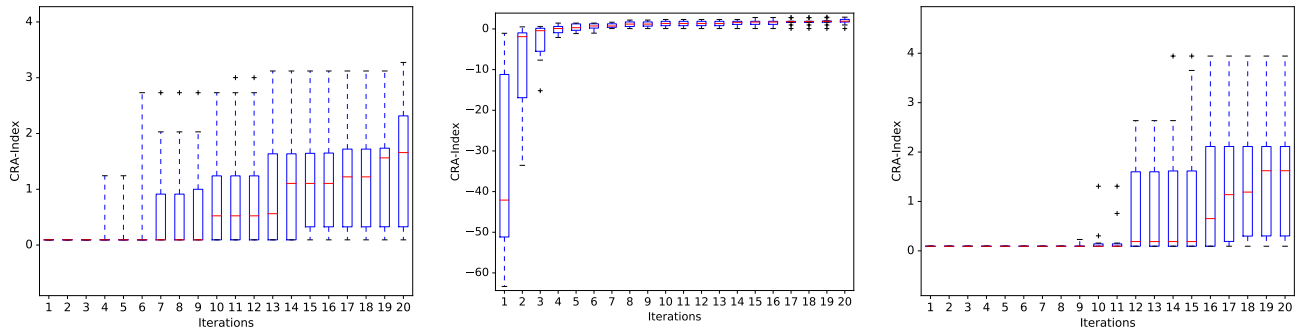


Figure 10: CRA of input model E depending on the selected initialization *allFeaturesInOneClass*, *oneClassPerFeature* or *mixed* (from left to left).

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.

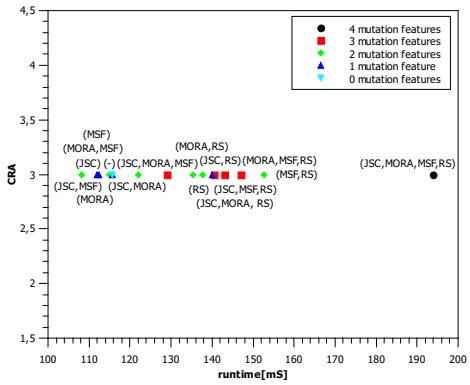


Figure 11: Input model A

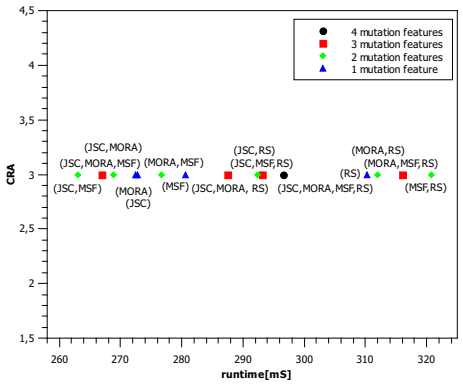


Figure 12: Input model B

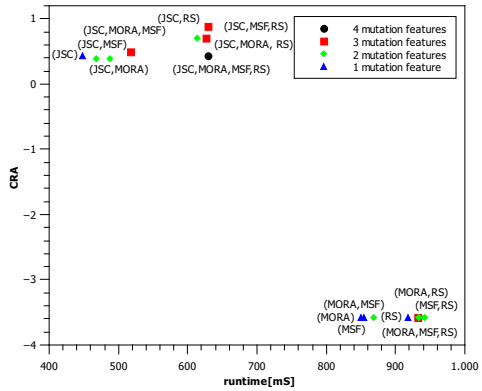


Figure 13: Input model C

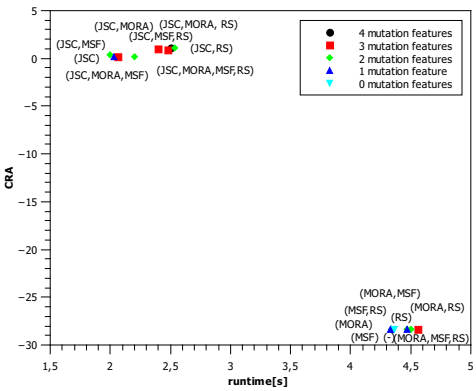


Figure 14: Input model D

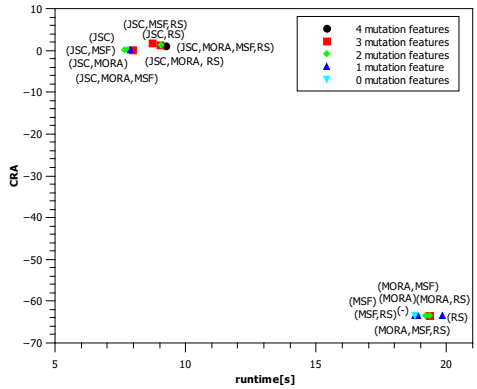


Figure 15: Input model E

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.

Solving the TTC'16 Class Responsibility Assignment Case Study with SIGMA and Multi-Objective Genetic Algorithms

Filip Křikava
Faculty of Information Technology
Czech Technical University
filip.krikava@fit.cvut.fr

Abstract

In this paper we describe a solution for the *Transformation Tool Contest 2016* (TTC'16) Class Responsibility Assignment (CRA) case study using SIGMA, a family of Scala internal *Domain-Specific Languages* (DSLs) that provide an expressive and efficient API for model consistency checking and model transformations. Since the Class Responsibility Assignment problem is a search-based problem, we base our solution on multi-objective genetic algorithms. Concretely, we use NSGA-III and SPEA2 to minimize the coupling between classes' structural features and to maximize their cohesion.

1 Introduction

In this paper we describe our solution for the TTC'16 Class Responsibility Assignment (CRA) case study [FTW16] using SIGMA [KCF14]. The goal of this case study is to find high-quality class diagrams from existing responsibility dependency graphs (RDG). The RDGs only contain a set of methods and attributes with functional and data relationships among them. The CRA problem is essentially about deciding where the different responsibilities in the form of class structural features (*i.e.* operations and attributes) belong and how objects should interact by using those operations [BBL10]. Since the design space of all possible class diagrams grows exponentially with the size of the RDG model [FTW16] (*i.e.* the number of structural features), the problem is hard to solve. However, a possible approximation could be found using search-based optimization techniques [CLV07]. Concretely, the use of multi-objective genetic algorithms seems to provide an efficient solution for the CRA problem as demonstrated by Bowman *et al.* [BBL10].

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

In this paper, we therefore present a solution to the CRA problem using SIGMA and multi-objective genetic algorithms. We use SIGMA to transform the input RDG diagram into a search-based problem which is then solved by a genetic algorithm. In the implementation we use NSGA-III and SPEA2 algorithms from the MOEA framework[1]. The MOEA Framework is a free and open source Java library for developing and experimenting with multi-objective evolutionary algorithms (MOEAs) and other general-purpose multi-objective optimization algorithms [Had16].

SIGMA is a family of Scala[2] internal DSLs for model manipulation tasks such as model validation, model to model (M2M), and model to text (M2T) transformations. Scala is a statically typed production-ready *General-Purpose Language* (GPL) that supports both object-oriented and functional styles of programming. It uses type inference to combine static type safety with a *"look and feel"* close to dynamically typed languages.

SIGMA DSLs are embedded in Scala as a library allowing one to manipulate models using high-level constructs similar to the ones found in the external model manipulation DSLs. The intent is to provide an approach that developers can use to implement many of the practical model manipulations within a familiar environment, with a reduced learning overhead as well as improved usability and performance. The solution is based on the *Eclipse Modeling Framework* (EMF) [SBPM08], which is a popular meta-modeling framework widely used in both academia and industry, and which is directly supported by SIGMA.

In this particular TTC'16 case study, the main problem is in solving an optimization problem rather than a transformation problem. SIGMA is therefore only used for straightforward transformation of input RDG models into optimization problems and problems' solutions back into class diagrams.

The complete source code is available on Github[3]. In the Appendices A and B we provide steps how to install it locally as well as how to run it on the SHARE environment.

2 Solution Description

The core of this case study is to transform a RDG model into a high-quality class diagram (*cf.* Figure 1).

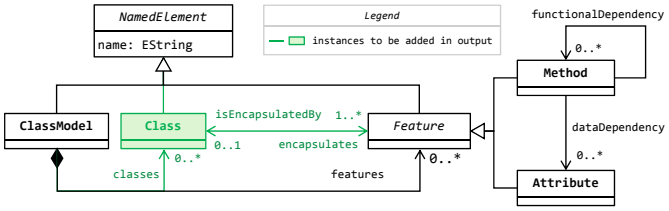


Figure 1: RDG and class model metamodel [FTW16]

To consider the quality of a class digram, two common software engineering metrics are used: *coupling* (the number of external dependencies) and *cohesion* (the number of internal dependencies). The two metrics can be further combined in one, single quality metric called *CRA-Index*, which simply subtracts the coupling from cohesion. The case study authors provide a set of utility functions that can compute all these metrics from a class digram instance and therefore we do not need to concern ourselves by their precise definitions.

The outline of the solution proposed in this paper is as follows:

[1] `http://www.moeaframework.org/`
[2] `http://scala-lang.org`
[3] `https://github.com/fikovnik/ttc16-cra-sigma`

1. Load the input RDG model from a given XMI file.
2. Transform the RDG model into MOEA problem instance.
3. Run the MOEA solver using either NSGA-III or SPEA2 algorithms.
4. From the possible solutions (which are part of a Pareto optimal front *cf.* below), select the one with the highest CRA-Index.
5. Transform the selected solution into class digram.
6. Save the resulting class diagram into XMI file.

2.1 Transformations

An optimization problem defines a search space, or the set of possible solutions together with one or more objective functions. In our case the search space spans are all the valid class diagrams that can represent a given RDG model. The objectives are: (1) to minimize coupling, and (2) to maximize cohesion.

The functions that compute coupling and cohesion ratios from a class diagram are part of the case study description. What remains is to find the way how to represent the RDG model as a vector of variables that can be used in a evolutionary algorithm to find a solution. We use a simple integer vector where the index corresponds to the feature index in the input RDG model and the value corresponds to the index of a class in the resulting class diagram. The range of each vector element is between 0 and the number of features -1 (since we use 0-based indexing). In the worst case, (*i.e.* one feature per class), this actually equals to the number of features. For example, a vector $(3, 5, \ldots)$ represents a solution in which first feature belongs to fourth class, second feature to sixth class, and so on and so forth. Figure 2 shows a further example of this representation on the example input/output model pair from the case description [FTW16].

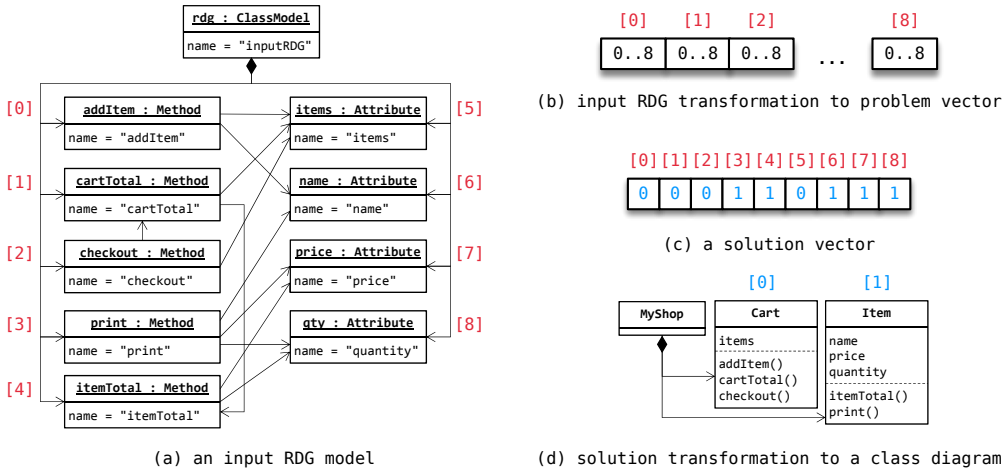


Figure 2: Example of the Solution

The advantage of this representation is that it can be easily mapped to MOEA decision variables. Also, each feature will always be assigned to (encapsulated by) some class. Therefore, the second validation constraint *all features provided in the input model must be encapsulated by a class*, will be always satisfied without any additional logic.

Concretely, in the MOEA framework, we have created a `Problem` class, called `CRAProblem`. The integer vector is used to define the decision variables, their types (*i.e.* integers) and bounds (*i.e.* $0 \ldots$ number of features $- 1$). The number of decision variables corresponds to the number of features in the input RDG model. The number of objectives is always two, the first one for coupling and the second one for cohesion. The method instantiating new solution instances looks

as follows:

```
override def newSolution() = {
  val s = new Solution(numVars, numObjs)
  (0 until numVars) foreach (x => s.setVariable(x, newInt(0, numVars - 1)))
  s // return the new instance
}
```

Next to providing a method to instantiate new instances of solutions for the problem, we need to also define the evaluation of a solution to compute the objectives. This involves two steps: (1) transforming the solution into a class diagram (2) using the provide `calculateCoupling` and `calculateCohesion` utility functions to compute the metrics. In code this is implemented as:

```
override def evaluate(s: Solution) = {
  val m = solutionToClassModel(initModel, s) // transformation
  s.setObjective(0,calculateCoupling(m))     // minimize coupling
  s.setObjective(1,-calculateCohesion(m))    // maximize cohesion
}
```

The negation of the cohesion ratio is due to the fact that MOEA only works on minimization problems and thus we need to negate the objective value to convert from maximization into minimization. The following is the code that does the transformation. This is the main code that uses SIGMA.

```
def solutionToClassModel(initModel: ClassModel, s: Solution) = {
  val m = initModel.sCopy // create a new model as a copy of the input one
  val v = EncodingUtils.getInt(solution) // get problem vector (v: Array[Int])
  // create new classes
  val classes = (0 to v.max) map (x => Class(name = s"Class $x"))

  // assignment
  v.zipWithIndex.foreach {
    case (cIdx, fIdx) => m.features(fIdx).isEncapsulatedBy = classes(cIdx)
  }

  // add non-empty classes
  m.classes ++= classes filter (x => !x.getEncapsulates.isEmpty)
  m
}
```

Finally, we define a new type, `Solver`, which is a function $RDG \rightarrow ClassDiagram$. The solver is responsible (1) to find the Pareto optimal front of all possible solutions (subject to solver configuration), and (2) to select the solution from that set which has the highest CRA-Index. The non-dominated, Pareto optimal front refers to optimal solutions whose corresponding vectors are non-dominated by any other solution vector [BBL10] and it can be found by MOEA `Executor`. For example using the NSGA-III algorithm, we find the non-dominated vector as:

```
new Executor().withProblemClass(classOf[CRAProblem], initModel)
              .withAlgorithm("NSGAIII")
              .withProperty("populationSize", 64)
              .withMaxEvaluations(10000)
              .run()
```

The individual solutions in this vector are first converted to the class model using the function `solutionToClassModel`. Then we use the given `calculateCRA` function to find the highest

CRA. To have a better chance to find a good solution, we run each algorithm 10 times (this starts the algorithm from 10 different random seeds). The properties of each algorithm are defined based on the suggestion by Bowman *et al.* [BBL10].

3 Evaluation

In this section we provide an evaluation of our solution following the categories given by the case study description. We leave the complexity and flexibility characteristics to be evaluated by reviewers. All the presented results are based on the NSGA-III algorithm which in all runs performed better than SPEA2. More results are provided on the github page.

Completeness & Correctness. The solution always converts a valid input RDG into a class model. The three constraints that were imposed by the solution description are solved as follows:

— *Every class must have a unique name.* The new classes are created in a loop that iterates over a number range. Part of the class name is the iteration variable and thus it must be always unique.

— *All features provided in the input model must be encapsulated by a class.* This has been already explained in the previous section. This is a property of the problem mapping we have chosen.

— *There cannot be any empty classes.* We explicitly filter out empty classes.

Optimality and Performance. The following table shows the cohesion and coupling rations, the resulting CRA-Index as well as the completion time from the SHARE environment[4]:

Input	Cohesion	Coupling	CRA	Time [s]
A	4	1	3	19.17
B	6.5	2.5	4	34.78
C	6.37	3.63	2.74	72.53
D	4.83	7.94	-3.11	300.49
E	7.38	17.99	-10.60	1110.74
F	9.85	44.74	-34.88	6289.75

References

[BBL10] Michael Bowman, Lionel C Briand, and Yvan Labiche. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. *Software Engineering, IEEE Transactions on*, 36(6):817–837, 2010.

[CLV07] Carlos Coello Coello, Gary B Lamont, and David A Van Veldhuizen. *Evolutionary algorithms for solving multi-objective problems*. Springer, 2007.

[FTW16] Martin Fleck, Javier Troya, and Manuel Wimmer. The Class Responsibility Assignment Case. In *Transformation Tool Contest 2016*, Vienna, 2016.

[Had16] David Hadka. MOEA Framework - A Free and Open Source Java Framework for Multiobjective Optimization. Version 2.10, 2016.

[KCF14] F. Krikava, P. Collet, and R. France. SIGMA: Scala Internal Domain-Specific Languages for Model Manipulations. In *Proceedings of 17th International Conference on Model-Driven Engineering Languages and Systems*, volume 8767, 2014.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, 2008.

[4]These are the initial results submitted for the competition. Improved result obtained after a better use of parallelism and better crossover and mutation ratios are available at `https://github.com/fikovnik/ttc16-cra-sigma`

A Install and Run Locally

The only requirements for running the solution is to have `git` and `sbt`[5] tools installed. To reproduce the benchmark simply execute these steps in a command line:

```
$ git clone \
  https://github.com/fikovnik/ttc16-cra-sigma
$ cd ttc16-cra-sigma
$ ./build.sh
$ ./run.sh
```

B Install and Run on SHARE

On the share environment we provide ready to be run solution. Simply log into the SHARE VM `remoteArchLinux64-TTC16_SIGMA` with `ttcuser/ttcuser` as user name/password and run the following:

```
$ cd ttc16-cra-sigma
$ ./run.sh
```

[5] simple-built-tool *cf.* `http://www.scala-sbt.org/`