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 provides 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 the 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 could be solved 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. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, 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. Furthermore, it is supported by the major integrated development environments bringing EMF modeling to other IDEs than traditionally Eclipse (the solution was developed in IntelliJ IDEA[3]).

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 to transform input RDG model into an optimization problem and to transform the problems' solutions into class diagrams.

The complete source code is available on Github[4]. In the Appendix 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 digram (*cf.* Figure 1).

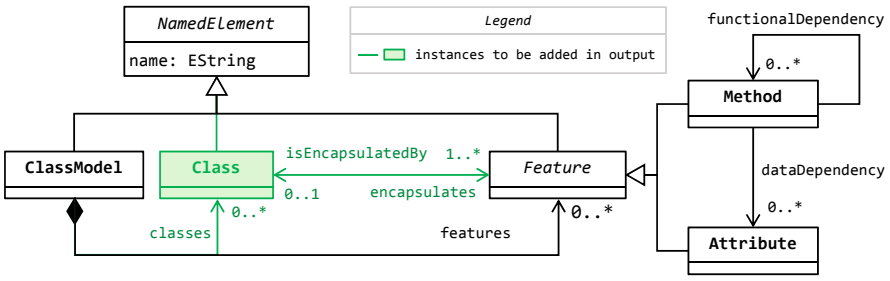


Figure 1: RDG and class model metamodel

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*,

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

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. Loads the input RDG model from a given XMI file.
2. Transforms the RDG model into MOEA problem instance.
3. Runs 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), selects the one with the highest CRA-Index.
5. Transforms the selected solution into class digram.
6. Saves 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 are all the valid class diagrams that can represent 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 bound of each vector element is bounded between 0 and the number of features -1 (since we use 0-based indexing). 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].

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

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 code that does the transformation is shown in Listing 1. This is the main code that uses SIGMA.

```
def solutionToClassModel(
  initModel: ClassModel,
  s: Solution) = {

  // create a new model as a copy
  // of the input one
  val m = initModel.sCopy
  // get problem vector (v: Array[Int])
  val v = EncodingUtils.getInt(solution)
  // 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
}
```

Listing 1: Solution transformation

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 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 Listing 1 and 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. The properties of each algorithm are defined based on the suggestion by Bowman *et al.* [BBL10].

The code is organized into three classes in `src/main/scala` folder:

— `CRAProblem` defines the CRA problem in terms of MOEA problem,
— `Solvers` preconfigures the two used algorithms for the finding the non-dominated solution vector, and finally
— `Main` that assembles the solution together into an executable application.

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. 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. The following table shows the cohesion and coupling rations as well as the resulting CRA-Index:

Input	Cohesion	Coupling	CRA
A	4	1	3
B	6.5	2.5	4
C	6.37	3.63	2.74
D	4.83	7.94	-3.11
E	7.38	17.99	-10.60
F	9.85	44.74	-34.88

Performance. The solution completion time from the SHARE environment for the input models is presented in the table below:

Input	Time [s]
A	19.17
B	34.78
C	72.53
D	300.49
E	1110.74
F	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 Science & Business Media, 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] Filip Krikava, Philippe Collet, and Robert 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, Valencia, 2014.

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

A Install and Run Locally

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

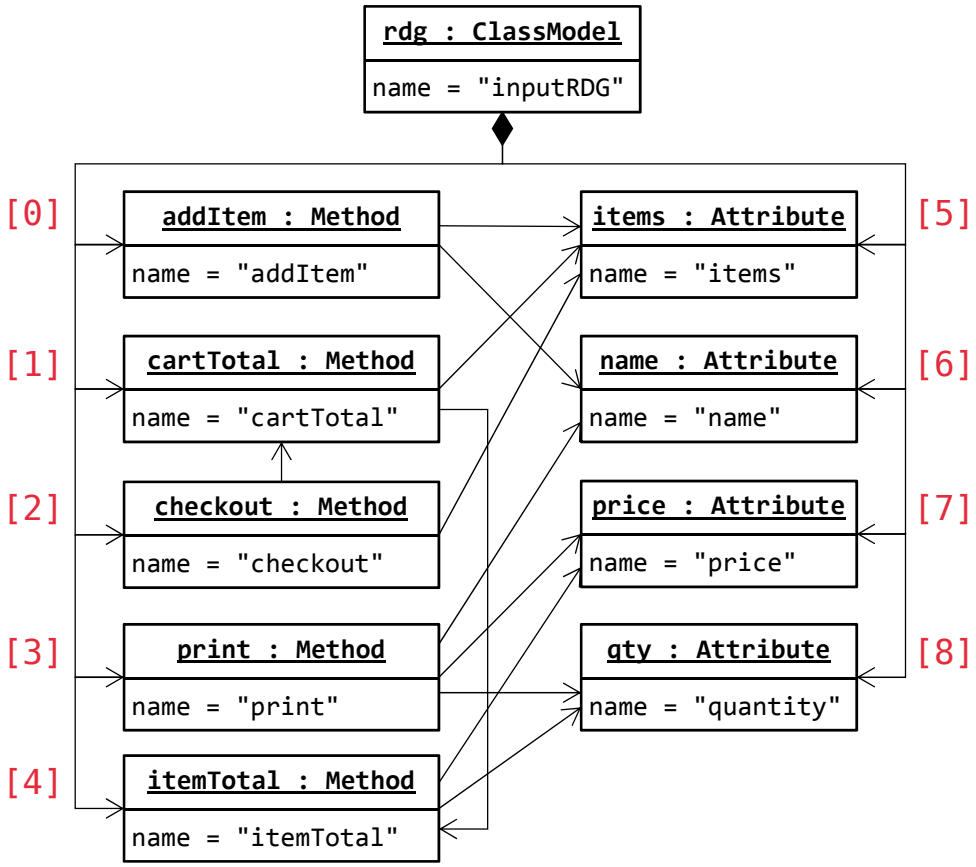
```
$ git clone \
  https://github.com/fikovnik/ttc16-cra-sigma
$ cd ttc16-cra-sigma
$ ./build.sh
$ ./run.sh
```

B Install and Run on SHARE

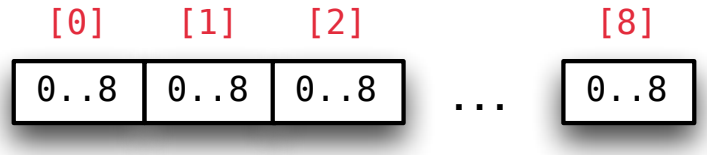
On the share environment we provide ready to be run solution. Simply log to the SHARE VM `remoteArchLinux64-TTC16_SIGMA` with `ttcuser/ttcuser` as user name/password and run the following:

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

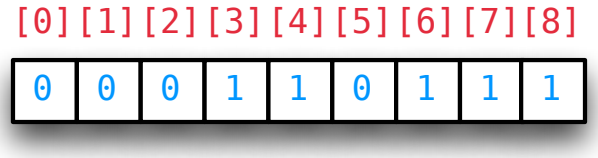
[5] simple-built-tool *cf.* `http://www.scala-sbt.org/`



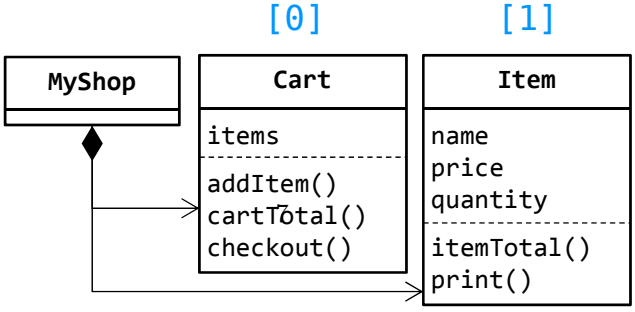
(a) an input RDG model



(b) input RDG transformation to problem vector



(c) a solution vector



(d) solution transformation to a class diagram

Figure 2: Example of the Solution