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

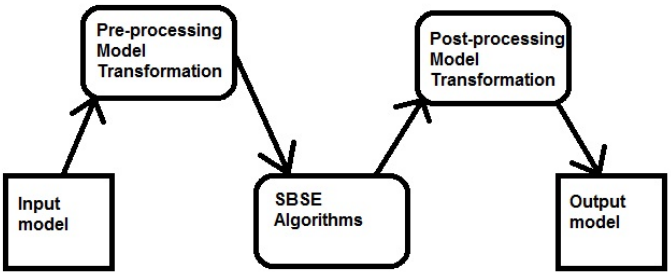

Figure 1: Integration of model transformations and SBSE

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.

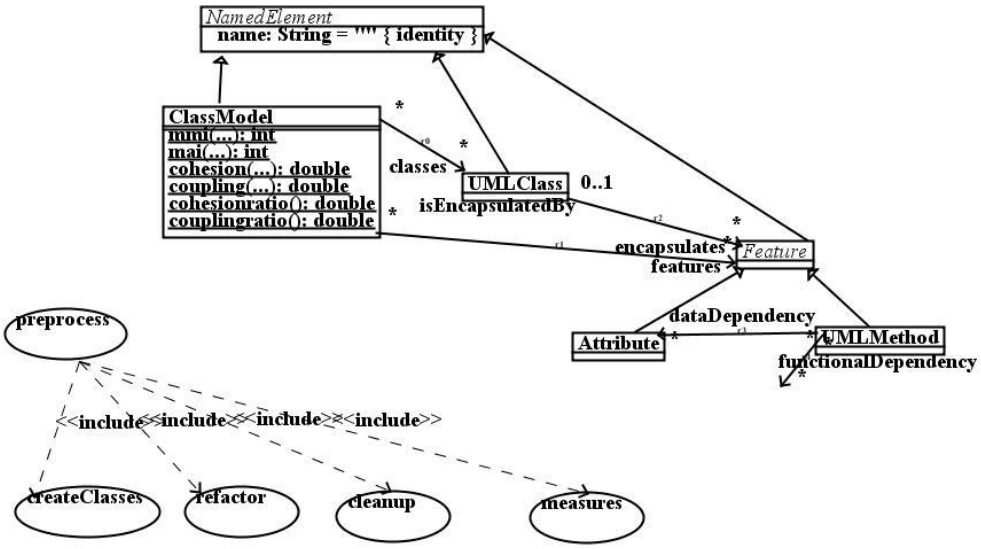

Figure 2: Pre-transformation (based on architectureCRA.ecore)

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.

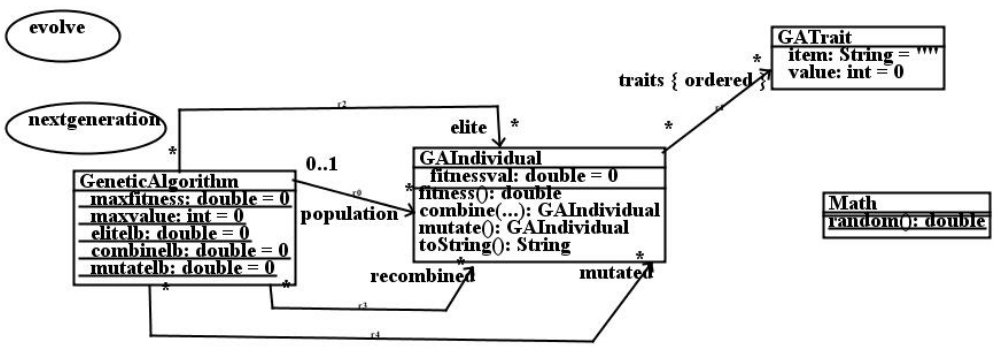

Figure 3: Genetic algorithm specification in UML-RSDS

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.