Solving the Class Responsibility Assignment Case with UML-RSDS

K. Lano, S. Kolahdouz-Rahimi, S. Yassipour-Tehrani
Dept. of Informatics, King's College London, Strand, London, UK

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 [2] is an endogenous transformation which aims to optimally assign attributes and methods to classes to improve a measure, CRA, 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).

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. 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 generic 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.

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

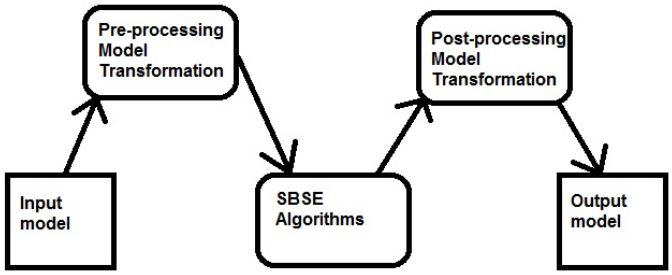

Figure 1: Integration of model transformations and SBSE

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.

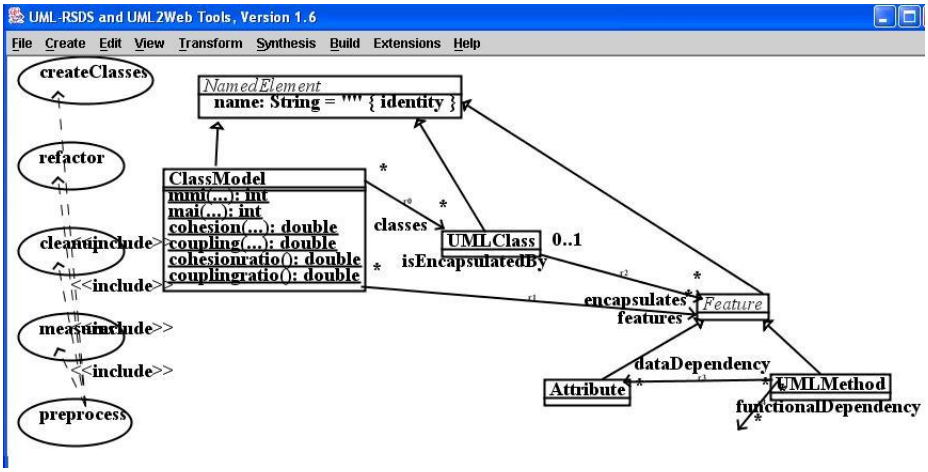

Figure 2: Pre-transformation (based on architectureCRA.ecore)

The rules for *createClasses* are as follows:

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

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

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

```
UMLMethod::
isEncapsulatedBy.size = 0 &
unencapdas = dataDependency->select( d | d.isEncapsulatedBy.size = 0 ) &
UMLMethod->forAll( m | m.isEncapsulatedBy.size = 0  =>
  m.dataDependency->select( a | a.isEncapsulatedBy.size = 0 )->size() <=
                unencapdas.size )   =>
        UMLClass->exists( c | c.name = "Class" + ( UMLClass@pre.size + 1 ) &
                 self : c.encapsulates & unencapdas <: c.encapsulates )

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

The first two rules locate all isolated features in a single class, *Class*0. The subsequent rules create classes for each 'chunk' of a method plus the attributes which are exclusively or primarily depended on by that method (Rule 4). Rules 3 and 5 enforce invariants which any reasonable class model should satisfy.

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*:

```
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

UMLClass::
a : encapsulates@pre & a->oclIsKindOf(Attribute) & c : UMLClass &
dependings = UMLMethod->select( f | f.dataDependency->includes(a) ) &
dependings->intersection(c.encapsulates@pre)->size()  >
      dependings->intersection(encapsulates@pre)->size()  =>  a : 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.

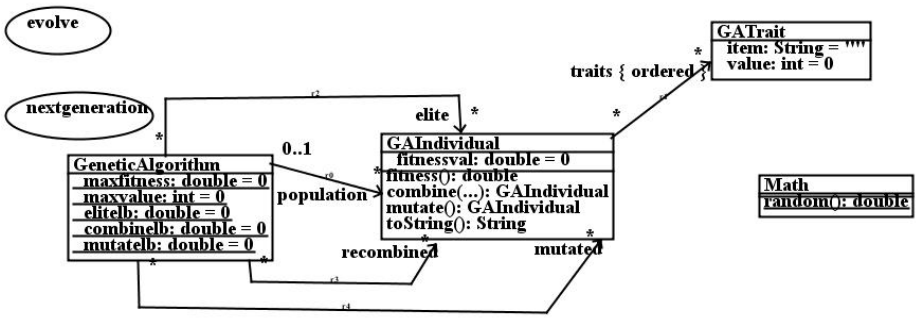

Figure 3: Genetic algorithm structure

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@pre.size > 500 & p : population@pre  =>
     population@pre->select( q | q /= p & q.fitnessval = p.fitnessval )->isDeleted()
```

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

Mutation consists of incrementing or decrementing the *value* of a randomly selected trait. Crossover takes place at a randomly-selected trait position.

To model the CRA problem in a GA, individuals represent possible assignments of features to classes, and have traits for each feature 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.:

$$Feature[item].isEncapsulatedBy \rightarrow includes(UMLClass.allInstances \rightarrow at(value))$$

$E[str]$ is the instance of entity type E with primary key value *str*. The fitness value is the CRA-index, computed using the definitions of [2]. The functions *dma* and *dmm* are *cached* to avoid repeated execution on the same inputs.

The $ga(iter : int)$ use case performs *initialise* to initialise the population with 3 copies of the model produced by *preprocess*, and 3 random individuals, then it iterates *evolve* and *nextgeneration* for *iter* times. Given a class model, a GeneticAlgorithm and GAIndividuals are generated by the rules:

```
ClassModel::
nclasses = UMLClass.allInstances.size  =>
    GeneticAlgorithm->exists( ga | ga.maxvalue = nclasses )

GeneticAlgorithm::
true => GAIndividual->exists( g |
   Feature->forAll( f | GATrait->exists( t | t.item = f.name &
      t.value = ( Math.random() * GeneticAlgorithm.maxvalue )->floor() + 1 &
      t : g.traits ) ) &
   g : population )

GeneticAlgorithm::
true => GAIndividual->exists( g |
   Feature->forAll( f | GATrait->exists( t | t.item = f.name &
      t.value = UMLClass.allInstances->indexOf(f.isEncapsulatedBy.any) &
      t : g.traits ) ) &
   g : population )
```

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

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

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 10 generations improves this to 0.494. For model D, the respective values are -20.83, -2.807, 0.369, and for model E -44, -20.37, -12.77. Incidently, the GA found an improved solution for the simple example of Figure 3 of [2]: this has CRA 1.58, and the method *addItem* in class *Item* instead of class *Cart*.

Test	*CRA after createClasses*	*CRA after refactor*	*CRA after* GA	*Execution time*
A	3	1.666	3	15ms + 47ms + 0ms
B	2.5	-1.5	2.75	32ms + 1s + 7ms
C	-3.917	-4.09	0.494	63ms + 8s + 16ms
D	-20.83	-2.807	0.369	338ms + 114s + 32ms
E	-44	-20.37	-12.77	1285ms + 1032s + 10ms

Table 1: Example test results

Table 2 shows the summary table completed for our solution.

Criteria	*Resolutions*
Correctness/completeness	All of the input models were converted to output models satisfying the constraints.
Optimality	We have obtained CRA values higher than those in the provided solutions.
Complexity	1 pd for MT, 1 pd for GA adaption + integration, 2pd for testing + optimisation
Flexibility	Alternative fitness functions (eg., giving higher weight to method-attribute dependencies) can be directly specified, as can stricter pruning of candidate solutions.
Performance	The GA was the most time-expensive element, but overall the execution time was practical.

Table 2: Solution evaluation table

All solution artifacts have been uploaded to SHARE (directory umlcra/umlcra), and are also 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.

For convenience, the generated Java code of the solution has been packaged as a jar file, and can be used as follows:

```
java -jar umlcra.jar inB.txt 8
```

where the 1st argument is the name of the model input file, and the 2nd is the number of GA generations to be performed. The output is written to out.txt (complete data) and to model.xmi (XMI format for the class model). The application can also be used interactively by running `java GUI`. With interactive execution, it is possible to apply refactor again during postprocessing to improve a GA-produced model. An example of console execution is:

```
C:\Documents\umlrsds16\output>java Controller inE.txt 4
...
Coupling ratio is:
47.41096664092987
Cohesion ratio is:
27.033169036845507
CRA is:
-20.377797604084364
Time for preprocess = 1342
Population size = 6 Maxfitness = -19.2892639427566
Population size = 18 Maxfitness = -16.838195865953228
Population size = 139 Maxfitness = -16.838195865953228
Population size = 128 Maxfitness = -14.144243113912228
Time for ga = 334947
Time for postprocess = 0
```

Input files are: inA.txt, inB.txt, etc. Example outputs are in outA.txt, etc.

Conclusions

It is clear that the level of individual attributes and methods is too low for the practical reverse-engineering or modularisation of large-scale software, and that appropriate groupings of features should be recognised as the units of organisation. For small-scale problems an approach based on individual features can produce acceptable results. We have shown that SBSE using genetic algorithms is able to enhance or produce results with better CRA values than MT-based refactoring used on its own. GA by themselves (with a random initial population) are also able to find good solutions, but the MT pre-processing accelerates the search. A drawback of the SBSE approach is that there is no rationale for the feature re-assignments – which the transformation rules provide for the MT approach. In addition, the outcome and execution time is subject to random variation. An alternative to genetic algorithms would be the use of a backtracking mode execution of refactoring rules – this approach has been used for game-playing software developed with UML-RSDS, however for this case study it would be difficult to define the backtracking critera to trigger rules to be undone. The CRA index is not necessarily the best measure for class diagram quality, and more subtle measures may be useful.

References

[1] SHARE site for solution: XP-TUe_UML-RSDS.vdi.

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

[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.