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 in 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 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 and are interested to see how we compare to search tools based on e.g. 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 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 few seconds and creating output models with a good CRA score. Further, the solution could be incrementalized with very 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[1] and SHARE[2].

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

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

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

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

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.

We created a heuristic to estimate this effect in a way that is computationally lightweight. First, we observed that

$$\begin{aligned} MAI(c_{ij}, c_{ij}) = MAI(c_i, c_i) + MAI(c_i, c_j) \\ + MAI(c_j, c_i) + MAI(c_j, c_j) \end{aligned}$$

and likewise for MMI. 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 shown in Figure 1.

The effect that this merge operation has on the coupling is more complex, which is why we apply an inexact heuristic. This heuristic simply assumes that the merge will not have any impact on other classes than c_i or c_j. Since these two classes are no longer present, the coupling between them is removed. Thus, we have an estimated effect on the coupling $\Delta Cou(c_i, c_j)$ as

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

We then estimate 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. 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`. This 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.

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

Figure 1: The impact of merging classes c_i and c_j on the Coherence Ratio

3.3 Creating an initial complete and correct solution

To create an initial complete and correct solution, we create a class for each feature in the class model, depicted in Listing 2.

```
foreach (var feature in classModel.Features)
{
  var featureClass = new Class()
  {
    Name = "C" + feature.Name
  };
  featureClass.Encapsulates.Add(feature);
  classModel.Classes.Add(featureClass);
}
```

Listing 2: Encapsulating every feature in its own class

3.4 Optimization

To conveniently specify the optimization, we first specify some inline helper methods to compute the MAI and MMI of two classes. The implementation for MAI is depicted in Listing 3, the implementation of MMI is equivalent.

```
var mai = new Func<IClass, IClass, double>((cl_i,
    cl_j) =>
  cl_i.Encapsulates.OfType<Method>()
    .SelectMany(m => m.DataDependency)
    .Intersect(cl_j.Encapsulates)
    .Count());
```

Listing 3: 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. This is depicted in Listing 4. The filter condition that the name of the first class shall be smaller than the name of the second class is only to make sure we create each tuple of different classes only once. For each such tuple, we also save all intermediate values that will be required to calculate the estimate ΔCRA.

```
var possibleMerges =
  from cl_i in classModel.Classes
  from cl_j in classModel.Classes
  where cl_i.Name.CompareTo(cl_j.Name) < 0
  select new
  {
    Cl_i = cl_i,
    Cl_j = cl_j,
    M_i = cl_i.Encapsulates.OfType<Method>().
        Count(),
    M_j = cl_j.Encapsulates.OfType<Method>().
        Count(),
    A_i = cl_i.Encapsulates.OfType<IAttribute>().
        Count(),
    A_j = cl_j.Encapsulates.OfType<IAttribute>().
        Count(),
    MAI_ii = mai(cl_i, cl_i),
    MAI_jj = mai(cl_j, cl_j),
    MAI_ij = mai(cl_i, cl_j),
    MAI_ji = mai(cl_j, cl_i),
    MMI_ii = mmi(cl_i, cl_i),
    MMI_jj = mmi(cl_j, cl_j),
    MMI_ij = mmi(cl_i, cl_j),
    MMI_ji = mmi(cl_j, cl_i)
  };
```

Listing 4: Identify possible merges

To determine which merge candidate is most promising, we further include two helper methods `atLeastOne` as $i \mapsto \max\{i, 1\}$ and `combinationCount` as $i \mapsto \max\{1, i(i-1)\}$ that will be used in several places in order to avoid division by zeroes.

To rate the candidates for merge operations, we assign an effect to them, which is exactly our aforementioned heuristic $\Delta CRA(c_i, c_j)$. The implementation is depicted in Listing 5. If any denominator is zero, the nominator will also be zero and we can safely avoid divisions by zero by ensuring that the denominator is at least one.

```
var prioritizedMerges = possibleMerges.Select(m
    =>
  new
  {
    Merge = m,
    Effect =
      // Delta of Cohesion based on data
          dependencies +
      (m.MAI_ii + m.MAI_ij + m.MAI_ji + m.MAI_jj)
          / atLeastOne((m.M_i + m.M_j) * (m.A_i +
          m.A_j)) - (m.MAI_ii / atLeastOne(m.M_i *
           m.A_i)) - (m.MAI_jj / atLeastOne(m.M_j
          * m.A_j))
      // Delta of Cohesion based on functional
          dependencies
      (m.MMI_ii + m.MMI_ij + m.MMI_ji + m.MMI_jj)
          / combinationCount((m.M_i + m.M_j)) - (m
          .MMI_ii / combinationCount(m.M_i)) - (m.
          MMI_jj / combinationCount(m.M_j)) +
      // Delta of Coupling between C_i and C_j
      (m.MAI_ij / atLeastOne(m.M_i * m.A_j)) + (m.
          MAI_ji / atLeastOne(m.M_j * m.A_i)) + (m
          .MMI_ij / atLeastOne(m.M_i * (m.M_j - 1)
          )) + (m.MMI_ji / atLeastOne(m.M_j * (m.
          M_i - 1)))
  }).OrderByDescending(m => m.Effect);
```

Listing 5: Sorting the merges by effect

Finally, we use this sorted set of possible merges and perform the merge operations. Here 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 a estimated effect ΔCRA below zero. This is depicted in Listing 6.

```
var nextMerge = prioritizedMerges.FirstOrDefault
    ();
var classCounter = 1;
while (nextMerge != null && nextMerge.Effect > 0)
{
  Console.WriteLine("Now␣merging␣{0}␣and␣{1}",
      nextMerge.Merge.Cl_i.Name, nextMerge.Merge.
      Cl_j.Name);
  var newFeatures = nextMerge.Merge.Cl_i.
      Encapsulates.Concat(nextMerge.Merge.Cl_j.
      Encapsulates).ToList();
  classModel.Classes.Remove(nextMerge.Merge.Cl_i)
      ;
  classModel.Classes.Remove(nextMerge.Merge.Cl_j)
      ;
  var newClass = new Class() { Name = "C" + (
      classCounter++).ToString() };
  newClass.Encapsulates.AddRange(newFeatures);
  classModel.Classes.Add(newClass);
  nextMerge = prioritizedMerges.FirstOrDefault();
}
```

Listing 6: Performing merge operations

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.

3.5 Serializing the resulting model

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

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

Listing 7: Saving the resulting model

Afterwards, the model can be saved simply by telling the repository to do so. This is depicted in Listing 7.

4 The potential of incrementality

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.

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 by a asymptotically better solutions.

This problem can be solved using implicit incrementalization approaches such as NMF Expressions. Indeed, our solution has to be modified only at a few places and the resulting code can be run incrementally. In particular, two more namespace imports are necessary, the definition of the helper functions `mai` and `mmi` have to be of type `ObservingFunc` and have to be called explicitly with an `Evaluate` method. Furthermore, the following line has to be added before Listing 6:

```
var prioritizedMergesInc = prioritizedMerges.
    AsNotifiable();
```

Listing 8: Activating incrementality

This switches on incremental execution using NMF Expressions. To use this incremental version of the query, the contents of Listing 6 also have to be adjusted to use the variable `prioritizedMergesInc` instead of `prioritizedMerges`. Our uploaded solution contains two different main classes where the difference can be seen very clearly.

Unfortunately, the dependency graph generated in this case is very large, since the mathematical computation of ΔCRA has a complex AST. NMF Expressions is not yet ready to detect that in this case, the best solution would be to compute this heuristic in batch mode and concentrate only on managing the search tree. We are working in this direction, but this approach is not yet implemented. Right now, NMF Expressions instead uses dynamic dependency graph that is a 1:1 mapping of the AST multiplied with the data going through. Hence, the dynamic dependency

	Input A	Input B	Input C	Input D	Input E
Correctness	•	•	•	•	•
Completeness	•	•	•	•	•
CRA-Index	2	1.667	1.541	-15.82	-27.83
Model Deserialization	192ms	185ms	163ms	160ms	155ms
Optimization	17.4ms	27.2ms	78.8ms	712.6ms	5,278.2ms
Model Serialization	11.2ms	11.6ms	12.4ms	10.0ms	10.8ms
Total Time	220.6ms	223.8ms	254.6ms	882.6ms	5,443.8ms

Table 1: Summary of Results for batch mode solution

	Input A	Input B	Input C	Input D	Input E
Correctness	•	•	•	•	•
Completeness	•	•	•	•	•
CRA-Index	1.167	1.667	0.782	-15.43	-31.47
Optimization	328ms	798ms	2,815ms	16,135ms	85,233ms

Table 2: Summary of Results for incremental solution

graph is very large so that its traversal is way more computational expensive than the savings due to the better asymptotical complexity.

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 batch mode and Table 2 for the solution in incremental mode. Each measurement is repeated 5 times. The times for model deserialization and -serialization are identical for both cases and are only depicted once.

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 input models provided, the optimization completes in a few seconds.

The models created by the incremental solution are a bit different due to a different ordering when there are multiple candidates with the same estimated effect to the CRA-index. The incremental solution shows much worse performance than the batch mode version and therefore, incrementality, at least using NMF Expressions, does not seem profitable in this case.

The solution consists of 110 lines of code (+3 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. 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. This is not obvious, but for this concrete example, we think it is acceptable.

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.

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