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:

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

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.

```
var repository = new ModelRepository();
var model = repository.Resolve(args[0]);
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.

```
var MAI = new Func<IClass, IClass, double>((cl_i, cl_j) =>
    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.

```
var dataDependingClasses =
   (from att in classIAttributes.Concat(classJAttributes)
    from meth in dataDependencies[att]
    select meth.IsEncapsulatedBy).Distinct();
var dataDependentClasses =
   (from meth in classIMethods.Concat(classJMethods)
    from dataDep in meth.DataDependency
    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.

```
var prioritizedMerges = (from cl_i in classModel.Classes where cl_i.Encapsulates.All(f => f is IAttribute)
   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, Effect = Effect(cl_i, cl_j)})
   .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).

```
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);
```

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.

```
classModel.Name = "Optimized␣Class␣Model";
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`