The EMFeR Solution to the TTC 2018 Software Mapping Case

Christoph Eickhoff
Kassel University
christoph@uni-kassel.de

Simon-Lennert Raesch
Kassel University
raesch@uni-kassel.de

Albert Zündorf
Kassel University
zuendorf@uni-kassel.de

1 Introduction

The Transformation Tool Contest 2018 Software Mapping Case [SMCase] asks for choosing implementation variants for computation requests and for mapping chosen implementations on appropriate hardware nodes.

This paper shows the EMFeR [EMFeR, ERZ18] solution to this case. The Github repositories for EMFeR is https://github.com/fujaba/EMFeR and the Github repository for this solution may be found at https://github.com/fujaba/EMFeRTTC18

2 From SDMLib to EMFeR reachability graphs

The Software Engineering Group at Kassel University works on SDMLib [SDMLib], a library supporting Story Driven Modeling [NZR13, SDMGuide]. SDMLib provides class models and graph transformations and *reachability graph computation*. Reachability graph computation has been proposed by Arendt Rensik in his Groove system [Groove]. Reachability grpah computation starts with an initial situation e.g. a set of requests for components with multiple alternative implementation variants that may be deployed on several alternative compute nodes. Next, reachability graph computation deploys a number of nondeterministic model (graph) transformations that evolve a given situation in different directions. For example, a transformation may choose `implementation_0i0` for `component_0` or alternatively `implementation_0i1`. After generating all possible successor situations, the reachability graph computation starts over and generates all possible successors of these successors, again and again. The process terminates when all reachable situations have been generated.

For some applications there may be different sequences of transformation applications that result in the same situation. To detect such cases, reachability graph computation deploys sophisticated hash key computation that allow to identify isomorphic situations, easily. In addition, reachability graph computation deploys an explicit isomorphism check to identify accidental hash key clashes.

In [ERZ16] we used the SDMLib reachability graph computation to attack the attributes and methods clustering case of the Transformation Tool Contest 2016. Our performance on that case was, well, below average. As each new situation is realized as a full copy of all objects that represent the old situation (and than some modifications of these objects), we ran out of memory after generating about 20.000 situations. To deal with this limitation, in [ERZ16] we optimized our reachability graph computation by deploying a metric that is used for prioritizing the set of new situations in order to extend the most promising situations first. This leads to some sort of hill climbing algorithm combined with a kind of taboo search as we backtrack out of explored "hills" and the isomorphism checks avoid to revisit explored situations again.

In addition to our experiences with the methods clustering case of the Transformation Tool Contest 2016 we have deployed reachability graph computation for solving computer game puzzles, e.g. [P+16].

From these experiences we identified two main areas for possible improvement of our reachability graph computation and to come up with our new reachability graph computation in *EMFeR - EMF extended by Reachability graphs* [EMFeR]:

In: A. Garcia-Dominguez, F. Krikava and G. Hinkel (eds.): Proceedings of the 10th Transformation Tool Contest, Marburg, Germany, 21-07-2017, published at http://ceur-ws.org

```
1  package uniks.ttc18
2  class ESolution {
3          refers EAssignment[] assignments
4  }
5  class EAssignment {
6          String requestName
7          String compName
8          String implName
9          String nodeName
10         refers EAssignment[] assignments
11 }
12 class EChoice {
13         refers EAssignment assignment
14         String resName
15 }
```

Listing 1: EMF Class Model

1. SDMLib reachability graph computation relies on graph transformations that are not available in all technical areas. For example [P+16] has been build in dot net and we had to build a new graph transformation engine to apply reachability graph computation to that example. Thus, in EMFeR we replaced graph transformations by transformations written in general purpose programming languages e.g. in plain old Java.

2. SDMLib reachability graph computation does a full cloning of the current situation each time a transformation is applied. For the Software Mapping Case of the Transformation Tool Contest 2018 this e.g. means that we copy all objects representing computation nodes and their properties again and again although these objects never change. Therefore, EMFeR deploys a lazy copying mechanism that copies only objects that have been modified or that refer to modified objects. Thus, e.g. objects for compute nodes and their properties are shared between multiple situations.

Caution: EMF contains relationships and bidirectional associations prevent lazy copying. If for example we use a bidirectional reference between an `Implementation` object and a compute node object, a new reference would modify both objects and thus both objects need to be copied. If the compute node objects are (EMF) contained in a *Hardware Model* object this containment is again implemented as a bidirectional reference causing the hardware model object to be cloned and in turn all (other) contained computation node objects need to be cloned, too. Thus, to leverage lazy copying, one must restrict the (EMF) model to unidirectional associations.

Altogether, EMFeR reachability graph transformations may be used with transformations written in plain old Java and the lazy copying mechanism allows EMFeR to work with large object models and to be still able to generate some 100.000 to some 1.000.000 situations. Thus, in this paper we use EMFeR to attack the Software Mapping Case of the Transformation Tool Contest 2018

3 The solution

The Software Mapping Case of the Transformation Tool Contest 2018 comes with a data model based on abstract syntax trees (AST). This AST based data model is enriched by application specific access methods. Thus, the data model provides a convenient problem specific API. This works fine until you try to debug your solution. In the variables view of your IDE, the references to sub objects are mostly hidden in an AST children list. This made working with the data model quite uncomfortable.

Anyhow, our EMFeR framework currently relies on EMF data models. Thus, we could not directly use the provided data model but created our own EMF based data model and forward and backward transformations from the provided data model into our data model and vice versa. Listing 1 shows our data model in Xcore notation [Xcore]. Basically, out data model comprises only an `ESolution` object that refers to a number of nested `EAssignment` objects. Each `EAssignment` object refers to the names of the addressed `Request` and `Component` and the name of the chosen `Implementation` and hardware `Resource` node. This suffices to represent a possible solution for the current mapping task. In order to validate the solution and to compute its *energy* usage, we transform our solution into the original data model and perform the validation there.

Listing 2 shows our solver implementation. After some initializations in lines 2 to 4, lines 5 to 13 setup EMFeR

```
1  public Solution solve(Root model) throws SolvingException {
2     EMFeRTrafos emFeRTrafos = new EMFeRTrafos(model);
3     ESolution initialSolution = Ttc18Factory.eINSTANCE.createESolution();
4     emFeRTrafos.createTopLevelAssignments(initialSolution);
5     EMFeR emfer = new EMFeR()
6        .withStart(initialSolution)
7        .withMetric(solution -> emFeRTrafos.getNumberOfSolvedIssues(solution))
8        .withTrafo("choose implementation",
9           solution -> emFeRTrafos.getImplementationChoices(solution),
10          (solution, choice) -> emFeRTrafos.applyImplementationChoice(solution, choice))
11       .withTrafo("assign node",
12          solution -> emFeRTrafos.getComputeNodeChoices(solution),
13          (solution, choice) -> emFeRTrafos.assignComputeNode(solution, choice));
14    int noOfStates = emfer.explore();
15    Solution bestSolution = emFeRTrafos.getDummySolution();
16    double bestObjective = Double.MAX_VALUE;
17    for (ReachableState state : emfer.getReachabilityGraph().getStates()) {
18       ESolution newSolution = (ESolution) state.getRoot();
19       Solution emferSolution = emFeRTrafos.transformPartial(newSolution);
20       if ( ! emferSolution.isValid()) {
21          continue;
22       }
23       double newObjective = emferSolution.computeObjective();
24       if (newObjective < bestObjective) {
25          bestSolution = emferSolution;
26          bestObjective = newObjective;
27       }
28    }
29    return bestSolution;
30 }
```

Listing 2: Calling EMFeR

for the computation of a reachability graph that explores possible solutions for the `model` at hand.

Line 6 provides the initial `ESolution` object to EMFeR. This initial `ESolution` object has already assignments for each top level request but no implementation and no computation node has been selected, yet. Line 7 provides a metric computation to EMFeR. Currently, we use a very simple metric that counts the number of decisions that have already been made. Thus, EMFeR will always expand the situation next, that already contains the most decisions. This basically results in a depth first search strategy for the solution space.

Lines 8 to 10 provides a model transformation to EMFeR that deals with the selection of alternative implementation variants. This is done in two steps. First, line 9 provides operation `getImplementationChoices` to EMFeR. This operations computes a set of `EChoice` objects (cf. Listing 1) where each `EChoice` object describe the selection of a certain `Implementation` variant for a certain `Component` addressed by a certain `EAssignment`. For each such `EChoice` object EMFeR calls operation `applyImplementationChoice` (on a lazy copy of the current situation). This creates a new situation where the corresponding `Implementation` has been assigned to the addressed `EAssignment` and in case of sub component requests new `EAssignments` has been created that yet wait for new choices to be made.

Similarly, lines 11 to 13 deal with choosing hardware `Resource` node objects for the deployment of `Implementations`. In line 12, operation `getComputeNodeChoice` computes set of `EChoice` objects that assign alternative hardware `Resource` node objects to `EAssignments`. Operation `assignComputeNode` does the assignment.

Line 14 of Listing 2 does the reachabiltiy graph computation or search space exploration. Operation `emfer.explore` terminates when all possible alternatives have been explored or after generating a default of 300.000 situations. (You may pass another limit for the number of explored situations as parameter.)

The loop from line 17 now iterates through all generated situations and line 19 transforms each situation

into the data model of the Software Mapping benchmark. This enables line 20 to check the chosen solution for validity. Lines 23 to 27 compare each valid solution with the best valid solution known so far and in case of an improvement, the new solution becomes the best solution. Finally, line 29 return the `bestSolution`.

4 Results

Table 1 shows the benchmarking result of EMFeR compared with the simple solution and the ILPDirect solution provided with the Software Mapping Case. After all, the EMFeR solution works correct for the simple cases but it does not perform better than the simple brute force solution coming with the Software Mapping Case. The problem is that we currently use a kind of depth first search strategy. This does not work well, if EMFeR e.g. chooses an implementation for the very first component and this implementation cannot be deployed to any hardware node. In that case, EMFeR iterates through all possible hardware assignments before it reconsiders the early implementation choice. This runs into a time out very easily.

One idea to improve our approach is to validate choices before trying them. This means, operation `getImplementationChoice` from line 10 of Listing 2 should not compute *EChoice* objects that use implementations that cannot be deployed on any hardware node. Unfortunately, we did not manage to ask the provided solution checking mechanisms coming with the Software Mapping Case to check partial solutions for validity. The provided solution checking needs to evaluate attribute computations that may refer to parent and child nodes in the abstract syntax tree and these have to exist in order to enable the computation. It is probably possible, to use a more relaxed checking mechanism that approximate the validity of implementation choices. Unfortunately, we ran out of time on this problem.

Even if we manage to improve our choice set computation, EMFeR is not well suited for the Software Mapping Case. EMFeR's reachability graphs keep all visited situations in memory in order to identify previously visited situations. This pays off, when a situation is visited again and we do not need to revisit all its successors. However, the solution for the Software Mapping Case computes sets of choices for one assignment after the other. Thus, it visits the search space in a strictly tree structured manner and it never visits the same situation twice. Accordingly, keeping old states in memory does not pay off in this case and it becomes just an overhead. So, we have little hope to win this benchmark case and this somehow demotivated us to put more effort into optimizing our search strategy. We are looking forward to more suited benchmarks next year.

Benchmark	Simple	Solution	EMFeR		ILPDirect	
	time ms	objective	time ms	objective	time ms	objective
0_trivial	3	226823.67	542	226823.67	139	226823.67
1_small	5	34620.20	57	34620.20	38	34620.20
2_small-many-hw	10	34620.20	99	34620.20	124	34620.20
3_small-complex-sw	10	-	163,158	-	1062	34620.20
4_medium	120,002	-	101,525	-	124,367	5467973.79
5_medium-many-hw	120,002	-	101,525	-	111,427	-

Table 1: Eliminate States

References

[EMFeR] EMFeR - EMF extended by Reachbility Graphs https://github.com/fujaba/EMFeR June, 2018.

[ERZ16] Christoph Eickhoff, Lennert Raesch, Albert Zündorf The SDMLib Solution to the Class Responsibility Assignment Case for TTC2016. TTC@ STAF, 27-32

[ERZ18] Christoph Eickhoff, Lennert Raesch, Albert Zündorf EMFeR: Model Checking for (EMF) Models Submitted to Models 2018, 14-19 October 2018 Copenhagen, Denmark

[Groove] GROOVE - GRaphs for Object-Oriented VErification http://groove.cs.utwente.nl June, 2018.

[NZR13] Ulrich Norbisrath, Albert Zündorf, Ruben Jubeh, Story Driven Modeling CreateSpace Independent Publishing Platform; Auflage: 1 (22. April 2013) ISBN-13: 978-1483949253

[SDMGuide] Ulrich Norbisrath, Albert Zündorf, Tobias George, Ruben Jubeh, Bodo Kraft, Sabine Sachweh Software Stories Guide 2017/7/31 https://kobra.bibliothek.uni-kassel.de/bitstream/urn:nbn:de:hebis:34-2017073153163/3/SoftwareStoriesNotationWhitePaper.pdf

[SDMLib] SDMLib - Story Driven Modeling Library *www.sdmlib.org* May, 2017.

[P+16] David Priemer, Tobias George, Marcel Hahn, Lennert Raesch, Albert Zündorf Using graph transformation for puzzle game level generation and validation ICGT - International Conference on Graph Transformation, 223-235, 2016

[SMCase] Sebastian Götz, Johannes Mey, Rene Schöne and Uwe Aßmann Quality-based Software-Selection and Hardware-Mapping as Model Transformation Problem *www.transformation-tool-contest.eu/TTC_2017_paper_4.pdf* May, 2017.

[Xcore] Xcore - Modeling for Programmers and Programming for Modelers. https://wiki.eclipse.org/Xcore