An NMF solution to the State Elimination case at the TTC 2017

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 State Elimination case at the Transformation Tool Contest (TTC) 2017 using the .NET Modeling Framework (NMF). The goal of this case was to investigate, to which degree model transformation technology may help to make the specification of state elimination more declarative and faster than the reference implementation in JFLAP for smaller models.

1 Introduction

State elimination is a well known technique in theoretical computer science to extract the regular expression represented by a given state machine: The regular expression represented by a state machine is extracted by eliminating states and simultaneously constructing regular expressions for the remaining states. This state elimination is repeated until only one initial and an end state is left and the regular expression can be obtained from the edge between these states.

In the scope of the Transformation Tool Contest 2017, this problem has been reified as a model transformation problem[GVPK17] in order to reason on the understandability, but also on the scalability of modern model transformation systems.

In this paper, we present a solution for the state elimination case using the .NET Modeling Framework (NMF, [Hin16]). However, the state elimination case does not fit either of the model transformation languages NTL [Hin13] or NMF Synchronizations [Hin15, HB17]. Both of these languages share the common assumption that it is a characterizing element of a model transformation that there is correspondence between elements of the source model and elements of some target model. Based on this simple assumption, NTL offers unidirectional, imperative model transformations meanwhile NMF Synchronizations is more declarative and supports also bidirectional and incremental transformations on top of NTL.

As both of these languages do not fit, we still have NMF Expressions[1], the incrementalization system used in NMF. This incrementalization system supports the incremental execution of *arbitrary* analyses, but rests on the assumption that the analysis is side-effect free.

Therefore, we present a solution using standard C# based on the model API generated for the given Ecore model using NMF. However, we tried to demonstrate the usage of declarative C#-parts that also reach a good degree of declarativeness.

Our solution is publicly available online[2] but not on SHARE because the used version of NMF requires at least Windows Vista which is not available in SHARE.

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] There is no dedicated publication yet, but usage examples can be found in [HH15].

[2] `https://github.com/georghinkel/state-elimination-mt`

The remainder of this paper is structured as follows: Section 2 presents our solution. Section 3 evaluates our solution with regard to performance and conciseness. Section 4 concludes the paper.

2 Solution

Our proposed solution is completely standard general-purpose code. However, the language features of C# make this code already very concise such that we believe there is no further language necessary to further reduce the size of the code.

As the very first step, the solution has to load the input model. For this, we need to transform the Ecore metamodel to the NMeta format used within NMF and generate code for it. Both steps can be done together using the tool `Ecore2Code` that ships with the NMF Nuget Package `NMF-Basics`.

The code generator is aware of multiplicities, containments and opposite references and generates the code appropriately. However, we manually refined the generated NMeta metamodel to set lower bounds of the involved attributes to 1. As a reason, NMF generates nullable types for attributes with lower bound 0 and they are much more cumbersome to use.

The actual deserialization of the input model is as straight forward as shown in Listing 1. We need to create a new repository and load the model into that repository.

```
1 var repository = new ModelRepository();
2 var transitionGraph = repository.Resolve(path).RootElements[0] as TransitionGraph;
```

Listing 1: Loading the input model

Our solution only solves the main task and extension task 1, due to time constraints. In the solution, we first select the initial state and create a new one, if the selected initial state has incoming edges. The implementation is depicted in Listing 2.

```
1  var initial = transitionGraph.States.FirstOrDefault(s => s.IsInitial);
2  if (initial.Incoming.Count > 0)
3  {
4    var newInitial = new State { IsInitial = true };
5    transitionGraph.Transitions.Add(new Transition
6    {
7      Source = newInitial,
8      Target = initial
9    });
10   initial = newInitial;
11 }
```

Listing 2: Creating a new initial state, if the initial state has an incoming state

The solution makes intensive use of the ability of C# to initialize objects inline. This syntax feature is also supported by NMF, at least for single-valued features. We also make use of the fact that NMF respects bidirectional references. Therefore, the assignments in Lines 7 and 8 of Listing 2 do not only set the `Source` and `Target` property of the newly created transition object, they also implicitly add the new transition to the `Incoming` and `Outgoing` collection for the respective states. For this to work, NMF generates special collection implementations for these two collections. This simplifies the manually written code.

Next, we select a final state. Because the logic for this is slightly more complex, we extracted it into a method whose implementation is depicted in Listing 3. At first, we obtain a list of all states that are currently set as final states. If this list only contains a single element, there is no need to change anything and we simply select the state as the new final state. If there are multiple states, we create a new final state and add transitions from the existing final states to that new state. Adding those transitions is done exactly in the same way as in Listing 2.

```
1  var finalStates = transitionGraph.States.Where(s => s.IsFinal).ToList();
2  if (finalStates.Count == 1 && finalStates[0].Outgoing.Count == 0)
3  {
4    return finalStates[0];
5  }
6  else
7  {
8    var newFinal = new State();
9    foreach (var s in finalStates)
10   {
11     transitionGraph.Transitions.Add(new Transition
12     {
13       Source = s,
```

```
      Target = newFinal
    });
  }
  transitionGraph.States.Add(newFinal);
  return newFinal;
}
```

Listing 3: Creating a new final state, if multiple final states exist

The next part of the solution is the removal of states. Before we describe the implementation of this step, we take a step back to review the algorithmics of the state elimination.

Roughly, removing a state implies to update $i \cdot o$ transitions where i is the number of incoming and o is the number of outgoing transitions.

If we converted the automaton to a simple automaton as suggested in the case description, this implies $i = n$ and $o = n$ where n is the number of states (which decreases as states are eliminated). This immediately yields an effort of $\Omega(n^3)$. For very large numbers of n, this is highly problematic. Therefore, we create transitions lazily to reduce the complexity. The example state machines are rather sparse, meaning that the average in and out degree of the states is low in comparison with the number of states.

Still, we have to create or update $i \cdot o$ transitions whenever we delete a state. If we create a transition, this raises the product $i \cdot o$ for later removed states. Therefore, it is reasonable to delete those states with a low product $i \cdot o$ first as we assume that they have the least effect on eliminating other states.

The implementation of this step is depicted in Listing 4. In the first line of this listing, we sort the states by the product $i \cdot o$ before starting to remove states. Sorting the collection can be done highly declarative in C# using the `OrderBy` query operator. Because .NET disallows modifications of a collection while it is iterated, we copy the ordered version into an array.

Eliminating a state, we first check whether there are any self-transitions for the state to be removed and join them. To make this joining more readable, we picked the C# query syntax to select the labels of all self-edges. If there is a self-edge with a non-empty label, we directly star it for the regular expression.

```
foreach (var s in transitionGraph.States.OrderBy(s => s.Incoming.Count * s.Outgoing.Count).ToArray())
{
  if (s == initial || s == final) continue;

  var selfEdge = string.Join("+", from edge in s.Outgoing
                                  where edge.Target == s
                                  select edge.Label);

  if (!string.IsNullOrEmpty(selfEdge)) selfEdge = string.Concat("(", selfEdge, ")*");

  foreach (var incoming in s.Incoming.Where(t => t.Source != s))
  {
    if (incoming.Source == null) continue;
    foreach (var outgoing in s.Outgoing.Where(t => t.Target != s))
    {
      if (outgoing.Target == null) continue;
      var transition = incoming.Source.Outgoing.FirstOrDefault(t => t.Target == outgoing.Target);
      if (transition == null)
      {
        transitionGraph.Transitions.Add(new Transition
        {
          Source = incoming.Source,
          Target = outgoing.Target,
          Label = incoming.Label + selfEdge + outgoing.Label
        });
      }
      else
      {
        transition.Label = string.Concat("(", transition.Label, "+", incoming.Label,
                                         selfEdge, outgoing.Label, ")");
      }
    }
  }

  s.Delete();
}
```

Listing 4: Removing states

Before we actually remove the state, we iterate through all of its incoming and outgoing transitions. For each pair of incoming and outgoing transition, we create a new transition that avoids the state to be deleted. The label of that new transition is the same as following the `incoming` edge to the state marked for deletion, then

making cycles in that state (only in case there actually are cycles) and then following the `outgoing` edge to the target state.

This iteration through every pairs of incoming and outgoing transitions can be regarded as imperative code, but still has many declarative elements. For example, we do not need to specify *how* the collections are iterated or filtered and neither do we specify how a matching shortcut transition is found. Though these elements only provide a very thin abstraction layer, we think that they make the code quite understandable.

Finally, we delete the state that we previously picked from the list of all states. Through the generated code of NMF, this operation boils down to a simple call to a `Delete` method. It will remove the state from the collection it is contained in, i.e. the collection of states in the transition graph. Further, it resets all references to that state[3]. Because transitions are independent of states in the metamodel with respect to the containment hierarchy, the incoming and outgoing transitions will still exist after the state has been deleted. However, after the deletion operation, they are no longer connected to the deleted state, but the `Source` and `Target` reference are simply empty, i.e. point to `null`.

We could go on and delete those transitions as they are no longer valid. However, in our experiments, we came to the conclusion that it is better to leave these transitions in there because the effort to delete them (removing an element from an ordered collection is an $O(n)$-operation!) is larger than the additional overhead they imply for the traversing incoming or outgoing transitions of other states.

Finally, we return the possible paths from the initial state to the target state. This can be done similarly as above. The implementation, this time as a one-liner, is depicted in Listing 5.

```
1 return string.Join("+", from edge in initial.Outgoing where edge.Target == final select edge.Label);
```

Listing 5: Calculating the overall regular expression

3 Evaluation

Our solution is very concise. Besides generated code for the metamodel and one line of metamodel registration, the entire solution consists of 102 lines, of which 31 lines are either blank or only contain braces.

Model	# Elements	Regex size	Time to transform (ms)	Correct	JFLAP (ms)
leader3_2	61	33	192	✓	90
leader3_3	166	95	193	✓	490
leader3_4	359	210	206	✓	4,370
leader3_5	672	398	207	✓	58,600
leader3_6	1,135	675	218	✓	461,640
leader3_8	2,631	1,571	298	✓	–
leader4_2	139	76	227	✓	140
leader4_3	630	354	219	✓	57,780
leader4_4	1,881	1,067	253	✓	4,786,580
leader4_5	4,492	2,558	395	✓	–
leader4_6	9,221	5,262	831	✓	–
leader5_2	315	172	197	✓	3,460
leader5_3	2,344	1292	286	✓	–
leader5_4	9,513	5,267	890	✓	–
leader5_5	28,544	15,833	5,277	✓	–
leader6_2	735	398	207	✓	143,120
leader6_3	8,248	4,487	739	✓	–
leader6_4	45,865	24,979	17,210	✓	–
leader6_5	173,194	94,408	395,616	✓	–
leader6_6	515,077	280,865	4,356,603	✓	–
leader6_8	2,886,813	–	–	–	–

Table 1: Execution times for the example input models

[3]NMF can also raise an event to notify clients that an element was removed from the containment hierarchy but this event is not raised in this case as no clients are subscribed to it.

The execution times for the test models are depicted in Table 1. The execution times have been recorded on an Intel i7-4710MQ clocked at 2.50Ghz on a system with 16GB RAM. For the largest model *leader6_8*, the solution ran out of memory. The recorded times include the initialization of NMF, initializing metamodels, loading model instances and computing the regular expression. It does not include serializing the generated regular expression to a file or testing the expression for the example inputs.

The times show a very good performance and scalability. For most of the models, the regular expression can be computed in less than a second. In particular, for the example model *leader4_4*, the largest that the reference solution in JFLAP is able to process, our solution is faster than JFLAP by more than three orders of magnitude. Only for the largest models, the transformation takes a lot of time.

4 Conclusion

In this paper, we discussed how the .NET Modeling Framework can be used to solve state elimination reified as model transformation task. Because there is no correspondence between source and target model required, the transformation languages in NMF are not well suited for this problem. Hence, our solution is essentially using general-purpose code, although the implementation of bidirectional references and automatically resetting references for deleted model elements makes the implementation more concise.

The performance results for the smaller models are very good. In particular, the solution is able to process all except for the very largest models.

References

[GVPK17] Sinem Getir, Duc Anh Vu, Francois Peverali, and Timo Kehrer. State Elimination as Model Transformation Problem. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*, CEUR Workshop Proceedings. CEUR-WS.org, July 2017.

[HB17] Georg Hinkel and Erik Burger. Change Propagation and Bidirectionality in Internal Transformation DSLs. *Software & Systems Modeling*, 2017.

[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.

[Hin13] Georg Hinkel. An approach to maintainable model transformations using an internal DSL. Master's thesis, Karlsruhe Institute of Technology, October 2013.

[Hin15] Georg Hinkel. Change Propagation in an Internal Model Transformation Language. In Dimitris Kolovos and Manuel Wimmer, editors, *Theory and Practice of Model Transformations: 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings*, pages 3–17, Cham, 2015. Springer International Publishing.

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