Yage Solution to the State Elimination as a Model Transformation Problem of the TTC'17

Christoph Eickhoff
University of Kassel
Software Engineering
christoph@uni-kassel.de

Simon-Lennert Raesch
University of Kassel
Software Engineering
lennert@uni-kassel.de

Philipp Kolodziej
University of Kassel
Software Engineering
philipp@uni-kassel.de

Abstract

This paper describes the approach taken to solve the State Elimination as a Model Transformation Problem of the transformation tool contest 2017. The graph engine used is a new competitor in the field of model transformation tools. Yage (Yet another graph engine[1]) was developed by Philipp Kolodziej for his thesis at the Software Engineering Research Group in Kassel.

1 Introduction

Yage is a new and minimalistic yet powerful graph engine, still undergoing major changes. The TTC'17 is a great opportunity to find inspiration for further development and recognize the need for improvements. The biggest strengths of Yage are its simplicity, the simple data structures and algorithms in its core and basically, that it represents a fresh start. With that comes one of its biggest weaknesses - the lack of certain features. Working on this transformation problem and therefore encountering new and different challenges will expose those weaknesses and help to guide further development on the Yage project.

2 Getting to work with EMF input data

In order for Yage to work the input data needs to be of a specific format, i.e. not EMF data. To achieve this a model to model transformation is done during the loading of the EMF data available in XMI format. Yage is based upon a simplified graph model consisting of nodes and edges, with nodes explicitly modeled as classes but edges being associations. Nodes have a list of attributes that can be used to either represent data or store meta information such as the type of the represented object. The general approach is very generic and focuses on solving graph engine tasks, i.e. rule execution for graph transformations. The class model is depicted in figure 1.

3 Rules

3.1 Yage Rule Visualization

The visual notation of the rule graph is based upon a color scheme and circles, representing nodes and arrows, representing uni-directional edges between nodes. The color scheme enables the visualization to combine several steps of the rule matching and application into a unified image. For the first step, finding a suitable set of nodes, blue circles and nodes are used. They represent objects that need to be present, in their depicted configuration,

[1]`https://github.com/fujaba/org.fujaba.graphengine`

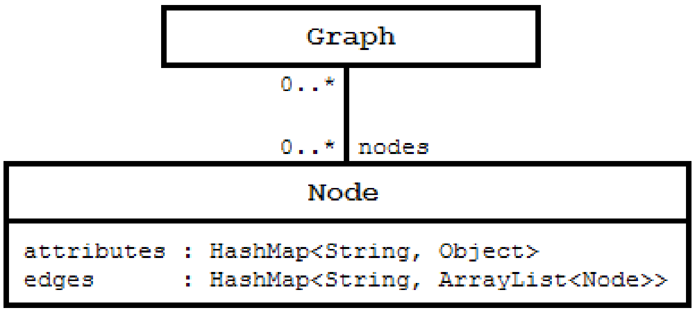

Graph
0..*
0..* nodes
Node
attributes : HashMap<String, Object>
edges : HashMap<String, ArrayList<Node>>

Figure 1: Generic Yage graph model

in the working graph. Lilac nodes and edges are considered just like blue circles in this first step. In the second step, the algorithm checks for nodes and edges depicted in red, meaning that those nodes and edges must not be present in the working graph for the rule to match. Only if a match has been found, nodes and edges of the color green are considered in addition to a second evaluation of the lilac objects. Nodes and edges of green color are created during this last step, nodes and edges in lilac have been matched in step one and are now being deleted from the working graph. The visualization thus makes the whole process of applying rules an easy to visually grasp concept, utilizing common colors and a simplified structure for nodes and edges. The deliberate decision to not use UML-like notations for objects abstracts the graph context from the more technical view of objects.

3.2 From algorithmic approach to rules and Algorithm objects

The solution to the problem consists of a multi-step algorithm. Once input data is loaded as a graph compatible to the Yage class model, the graph is then enhanced using a set of graph transformation rules. A new initial state is created and additional states are added to the this newly created state. Analogous a new final state is created and additional states are added again. Following this initial creation of new states, the next step is to repeatedly eliminate states. The algorithm chooses a state and adds an *eliminate* flag to it's attributes list. Additionally a source state is chosen and marked using a *current* flag. For each edge from the selected eliminate state a new edge leading to thats edge's target state is created and the state is flagged as *used.* Once all states reachable via the eliminate state have been flagged as used, these flags are removed, the current flag is removed and replaced by a *past* flag. This step is repeated for all applicable source (*current*) states per selected eliminate state. Once this process of merging all edges that lead via the node to be deleted, the node is removed and all flags are reset. As this process is repeated all possible combinations are considered. While the algorithmic approach is rather simple (see an incomplete pseudo-code implementation approach in listing 1) transforming this approach to a rule based one was not as straightforward. There are eight different cases for rerouting edges that involve the state to be removed, thus requiring a total of 20 rules for the solution to the case. In addition a couple of new features were introduced to Yage, namely the option to name edges selected in a previous rule for later referencing as well as the possibility to combine graph transformation rules to algorithms, with the possibility to repeatedly apply some rules or sequences of rules before continuing with the next rule in the algorithm. This is a very powerful approach similar to the Fujaba Activity Diagrams [Fujaba]. The transformation algorithm can also easily be serialized or deserialized from and to JSON. This way the complete algorithm can be exported and imported just like basic graphs and transformation rules. As these features are quite new, the syntax and especially the visualization have not been completely redesigned to be easily understandable. As can be seen in figure 1 and listing 2 naming edges isn't very intuitive yet. Implementing the solution to this problem has lead to further exploration of possible ways to define Yage's graph transformation tools and new ways of designing them will be a future task. Creating algorithms from graph transformation rules is on the opposite already a very intuitive way of creating solutions to more complex scenarios that involve multiple rules that are easier created as parts. It's syntax can be seen in listing 3.

Listing 1: Pseudo-code for algorithmic approach

```
loadGraphData();
prepareGraphData();
//start eliminating states
for(State delState : allStates){
        delState.flag("eliminate");
        for(State sourceState : allStates){
        if(delState != sourceState){
                sourceState.flag("current");
                for(Edge outgoing : delState.outgoingEdges){
                targetState = outgoing.targetState;
                new Edge(sourceState, targetState);
                targetState.flag("used");
                ...
            }
        }
    }
}
```

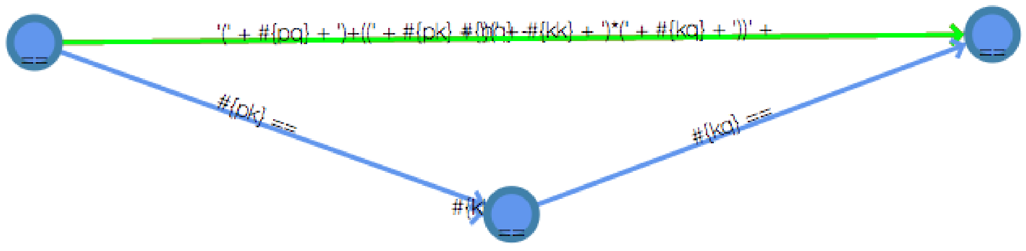

'(' + #{pq} + ')+((' + #{pk} #{}(h)+ #{kk} + ')*(' + #{kq} + '))' +
==
#{pk} ==
#{kq} ==
#{k
==
==

Figure 2: Prepare elimination of state (with pp, pk, kk, kp)

Listing 2: Prepare elimination of state (with pp, pk, kk, kp)

```
PatternGraph gtr =
        new PatternGraph("prepare␣elimination␣of␣state␣(with␣pp,␣pk,␣kk,␣kp)");
PatternNode p = new PatternNode("#{current}␣&&␣!(#{used})").addPatternAttribute(
        new PatternAttribute().setAction("+").setName("used").setValue(true));
PatternNode k = new PatternNode("#{eliminate}");
gtr.addPatternNode(p, k);
p.addPatternEdge("-", "#{pp}", p);
p.addPatternEdge("==", "#{pk}", k);
k.addPatternEdge("==", "#{kk}", k);
k.addPatternEdge("==", "#{kp}", p);
p.addPatternEdge(
        "+", "'((␣+␣#{pp}␣+␣')*((␣+␣#{pk}␣+␣')('␣+␣#{kk}␣+␣')*('␣+␣#{kp}␣+␣'))*)*'",
            p);
```

Listing 3: Creation of several Algorithm objects

```
Algorithm stateCaseTTC2017 = new Algorithm("TTC␣2017␣State␣Case");

Algorithm eliminateState = new Algorithm("eliminate␣state");
Algorithm handleSourceNode = new Algorithm("handle␣source␣node");
Algorithm redirectRoute = new Algorithm("redirect␣route");

stateCaseTTC2017.addAlgorithmStep(getStateCasePreparationAlgorithmTTC2017());
stateCaseTTC2017.addAlgorithmStep(eliminateState, true);

eliminateState.addAlgorithmStep(getMarkStateForEliminationPattern());
eliminateState.addAlgorithmStep(handleSourceNode, true);
handleSourceNode.addAlgorithmStep(getMarkWithCurrentPattern());
handleSourceNode.addAlgorithmStep(getMarkFallbackWithCurrentPattern());
handleSourceNode.addAlgorithmStep(redirectRoute, true);
redirectRoute.addAlgorithmStep(getPrepareStateWithPqPkKkKqPattern(), true);
redirectRoute.addAlgorithmStep(getPrepareStateWithPkKkKqPattern(), true);
redirectRoute.addAlgorithmStep(getPrepareStateWithPqPkKqPattern(), true);
redirectRoute.addAlgorithmStep(getPrepareStateWithPkKqPattern(), true);
redirectRoute.addAlgorithmStep(getPrepareStateWithPpPkKkKpPattern(), true);
redirectRoute.addAlgorithmStep(getPrepareStateWithPpPkKpPattern(), true);
redirectRoute.addAlgorithmStep(getPrepareStateWithPkKkKpPattern(), true);
redirectRoute.addAlgorithmStep(getPrepareStateWithPkKpPattern(), true);
handleSourceNode.addAlgorithmStep(getUnmarkCurrentPattern());
handleSourceNode.addAlgorithmStep(getRemoveMarksPattern(), true);
eliminateState.addAlgorithmStep(getEliminateMarkedStatePattern());
eliminateState.addAlgorithmStep(getUnmarkPastPattern(), true);
```

4 Results

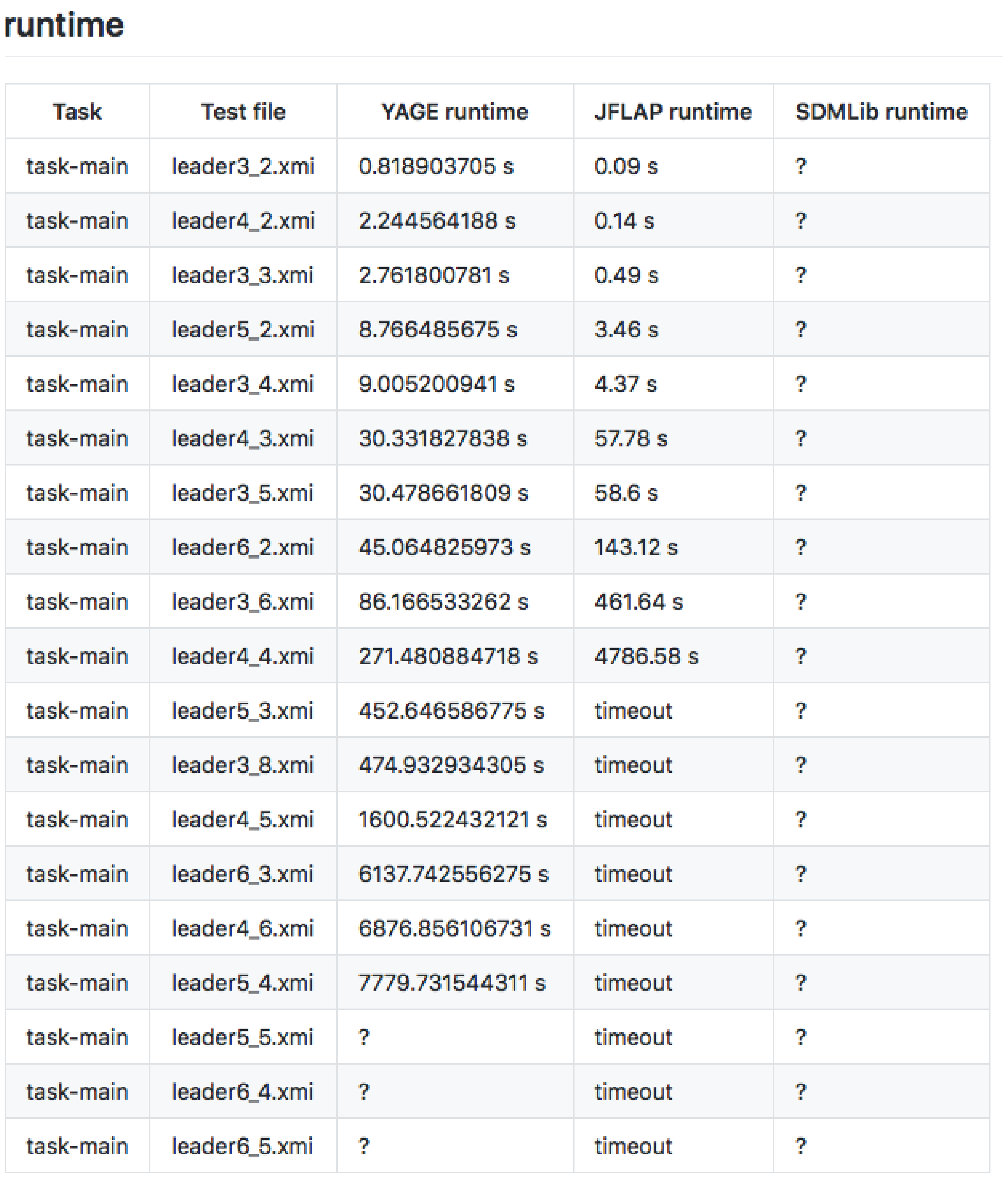
The results of the Yage transformations on the input data calculations can be seen in figure 3. We were able to successfully handle six of the input data files that the example JFLAP [JFLAP] implementation timed out on.

References

[JavaStringPool] http://docs.oracle.com/javase/specs/jls/se7/html/jls-3.html#jls-3.10.5

[JFLAP] http://www.jflap.org

[Fujaba] Ulrich Nickel, Jrg Niere, and Albert Zndorf. 2000. The FUJABA environment. In Proceedings of the 22nd international conference on Software engineering (ICSE '00). ACM, New York, NY, USA, 742-745. DOI=http://dx.doi.org/10.1145/337180.337620



runtime

Task	Test file	YAGE runtime	JFLAP runtime	SDMLib runtime
task-main	leader3_2.xmi	0.818903705 s	0.09 s	?
task-main	leader4_2.xmi	2.244564188 s	0.14 s	?
task-main	leader3_3.xmi	2.761800781 s	0.49 s	?
task-main	leader5_2.xmi	8.766485675 s	3.46 s	?
task-main	leader3_4.xmi	9.005200941 s	4.37 s	?
task-main	leader4_3.xmi	30.331827838 s	57.78 s	?
task-main	leader3_5.xmi	30.478661809 s	58.6 s	?
task-main	leader6_2.xmi	45.064825973 s	143.12 s	?
task-main	leader3_6.xmi	86.166533262 s	461.64 s	?
task-main	leader4_4.xmi	271.480884718 s	4786.58 s	?
task-main	leader5_3.xmi	452.646586775 s	timeout	?
task-main	leader3_8.xmi	474.932934305 s	timeout	?
task-main	leader4_5.xmi	1600.522432121 s	timeout	?
task-main	leader6_3.xmi	6137.742556275 s	timeout	?
task-main	leader4_6.xmi	6876.856106731 s	timeout	?
task-main	leader5_4.xmi	7779.731544311 s	timeout	?
task-main	leader5_5.xmi	?	timeout	?
task-main	leader6_4.xmi	?	timeout	?
task-main	leader6_5.xmi	?	timeout	?

Figure 3: Yage test results