Yage Solution to the Family to Persons Case 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 Family to Persons Case 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

2 Getting to work with EMF

The family to persons case is based on an EMF formatted data structure which is heavily modified during runtime. While prior years cases were also made available in EMF format, it was easily possible to load and then transform this data into a format usable by non-EMF-based graph engines, this is not the case with this years data. As a lot of modifications are applied to the initial data structure during runtime, i.e. during the benchmarking of the graph engine's approach to the case, it is not easily possible to apply EMF based restructuring of the data to our engines graph model. In order to solve this, we had to recreate the classes responsible for the runtime modifications, so model changes were applied to our own engines working graph. To make this as transparent as possible and to be able to adjust to changes more quickly, a subclass of the helper class responsible for applying changes to the working graph was created, overwriting the functions responsible for working with the graph. As Yage does not generate classes but works on a generic set of classes, the class and thus data structure differs greatly from the one given in the case's description, as can be seen in figure 1, depicting the given class structure and figure 2 of the generic Yage graph model.

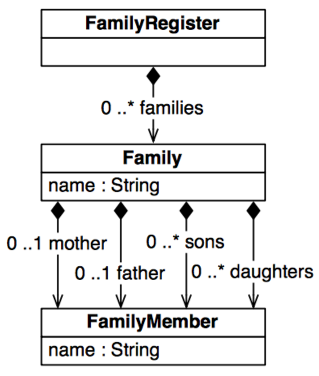
3 Rules

The necessary rules to transform the input data were written in the Yage syntax as can be seen in listing 1. To have a more visually appealing as well as descriptive view of the rules, Yage implements an HTML exporter that utilizes Alchemy.js [Alchemy.js] to plot the rule and working graphs.

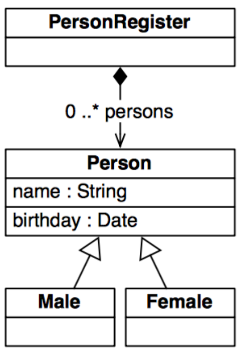
3.1 Yage Rule Visualization

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,

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



(a) Families metamodel



(b) Persons metamodel

Figure 1: Class model from the TTC case description

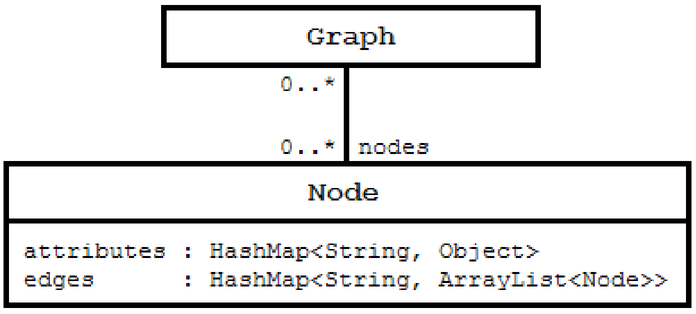

Graph
0..*
0..*
nodes
Node
attributes : HashMap<String, Object>
edges : HashMap<String, ArrayList<Node>>

Figure 2: Generic Yage graph model

blue circles and nodes are used. They represent objects that need to be present, in their depicted configuration, 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 Family To Person Rules

The implementation of the rules is straightforward and based on the case's description. PatterNode objects are graph nodes to be matched with the working graph's nodes. As nodes and edges are both Yage node objects, all explicit data is modeled via attributes, even a nodes type. Strings are nodes as well, modeled in a similar fashion as is done by the Java String pool implementation [JavaStringPool]. A visual representation of the listed code is shown in figure 3. Each circle represents a PatternNode object to be matched. Blue circles are nodes that need to be matched for the rule to be applied. Negative nodes, i.e. nodes that must not be present in the working graph are colored orange and connected with orange arrows, representing references, implemented as PatternEdge objects. Nodes to be added after a successful match of the rule are represented as green circles. To define whether a node or an edge is to be created, matched, not to be matcher or destroyed, Actions are set on the PatternEdge and PatternNode objects. These Action modifiers are defined as a String containing either == (to be matched), != (not to be matched), - (to be destroyed) and + (to be added). An Action for a PatternEdge that must not be present in the working graph for the rule to be applied is set in line 15. This example makes sure that no Person object with a given name and surname is referenced in the PersonRegister object. After the rule is successfully matched a node is created and associated with the name and surname nodes as well as the PersonRegister node. This rule thereby solves the task of creating persons from family members. It is repeated four times for combinations of daughters, sons, mothers and fathers with their respective gender, as can be seen in lines 21 and 27 of listing 1.

PatternEdge objects and PatternNode objects are stored in a PatternGraph object thus representing a complete graph transformation rule. In listing 1 the rule object is named patternGraph and assembled in line 12ff.

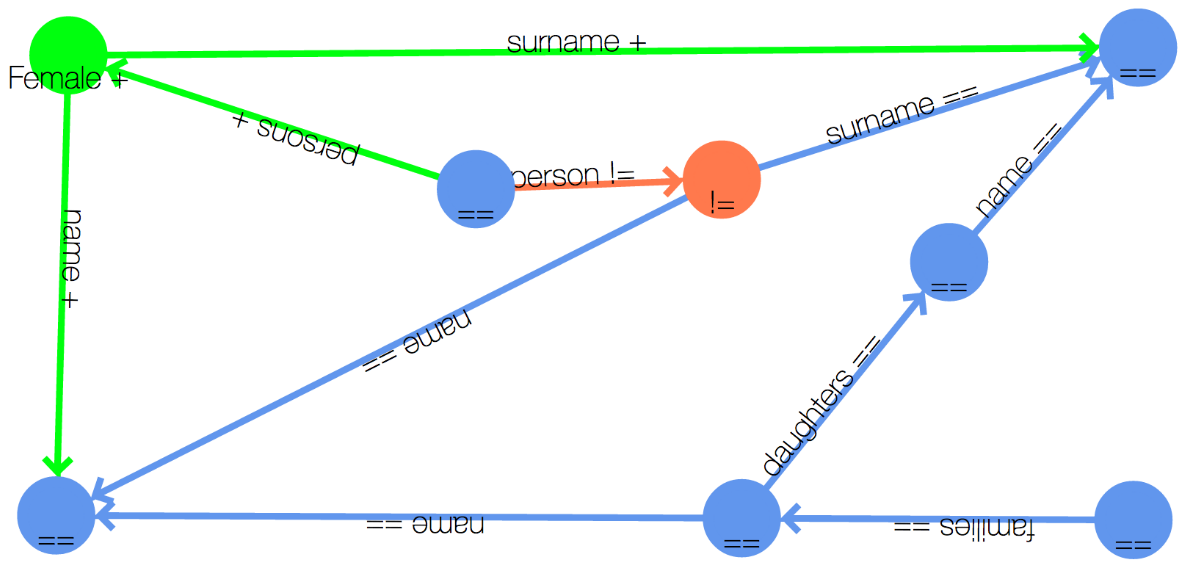


Figure 3: Family to person rule: Female from daughter

Listing 1: Family to person rule: Female from daughter

```
1     PatternNode familyRegister = new PatternNode();
2     PatternNode family = new PatternNode();
3     PatternNode familyMember = new PatternNode();
4     PatternNode name = new PatternNode();
5     PatternNode surname = new PatternNode();
6     PatternNode person = new PatternNode();
7     PatternNode personRegister = new PatternNode();
8     PatternNode newPerson = new PatternNode();
9
10    PatternGraph patternGraph = new PatternGraph("FamilytoPersonRule-" + FEMALE + "-"
          + DAUGHTER);
11
12    patternGraph.addPatternNode(family, familyMember, name, surname, person,
         newPerson, personRegister, familyRegister);
13
14    person.setAction("!=");
15    personRegister.addPatternEdge("!=", "person", person);
16
17    familyRegister.addPatternEdge(FAMILIES, family);
18    family.addPatternEdge(NAME, name);
19    familyMember.addPatternEdge(NAME, surname);
20
21    family.addPatternEdge(DAUGHTER, familyMember);
22
23    person.addPatternEdge("surname", surname);
```

```
        person.addPatternEdge("name", name);

        newPerson.setAction("+");
        newPerson.setPatternAttribute("+", Node.TYPE_ATTRIBUTE, FEMALE);

        newPerson.setPatternAttribute("+", BIRTHDAY, BIRTHDAY_EDEFAULT.toString());

        newPerson.addPatternEdge("+", "surname", surname);
        newPerson.addPatternEdge("+", "name", name);
        personRegister.addPatternEdge("+", "persons", newPerson);
```

3.3 Person To Family Rules

In addition to the Family To Person transformation the Person To Family transformation was implemented in a similar fashion. The code in listing 2 relates to the visualization of the rule in figure 4. Both rules thus require less then 50 lines of code, are of a readable format and can be visualized in an easy to understand, abstract format. This second rule (-set) is basically an inversion of the Person To Family rules. Given the presence of a node with links to the PersonRegister node, a name and surname node, and no corresponding node with a link to that name we then create a Family and FamilyMember node linking it to the existing nodes representing name, surname and FamilyRegister.

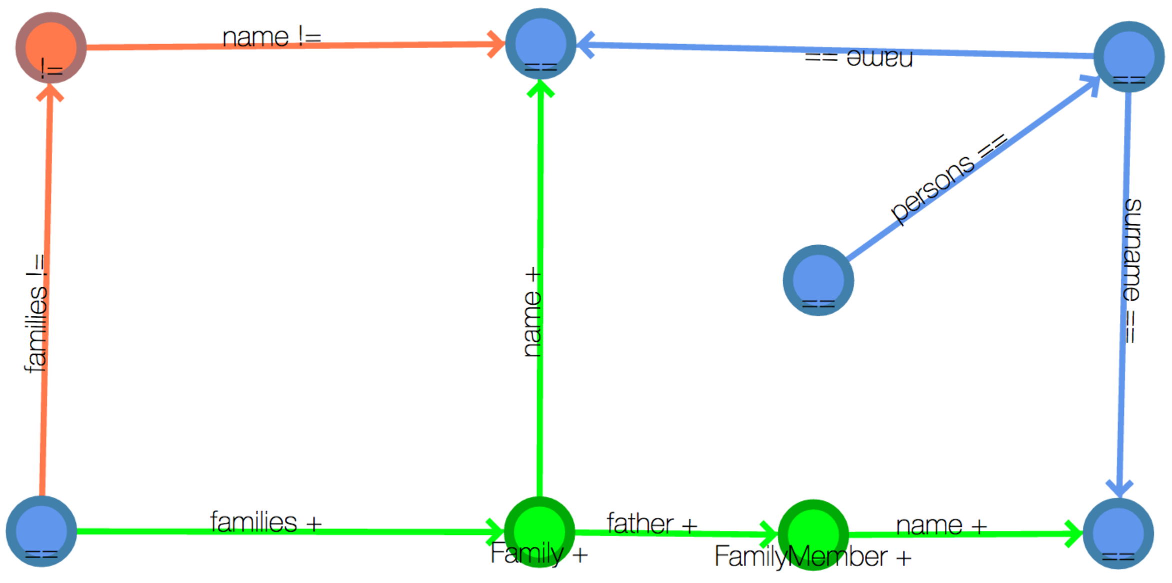


Figure 4: Person to family rule: Family and family member from male

Listing 2: Person to family rule: Family and family member from female

```
        PatternNode familyRegister = new PatternNode();
        PatternNode family = new PatternNode();
        PatternNode familyMember = new PatternNode();
        PatternNode name = new PatternNode();
        PatternNode surname = new PatternNode();
        PatternNode person = new PatternNode();
        PatternNode personRegister = new PatternNode();
        PatternNode newFamilyMember = new PatternNode();
        PatternNode newFamily = new PatternNode();

        PatternGraph patternGraph = new PatternGraph("PersonToFamilyRule" + DAUGHTER +
            FEMALE);
```

```
    patternGraph.addPatternNode(name, surname, person, personRegister, familyRegister
        , family, newFamily, newFamilyMember);

    personRegister.addPatternEdge(PERSONS, person);

    familyRegister.addPatternEdge("!=", FAMILIES, family);
    family.setAction("!=");
    family.addPatternEdge("!=", NAME, name);
    family.setAttributeMatchExpression(
            "#{" + Node.TYPE_ATTRIBUTE + "}=='" + FAMILY + "'");

    person.addPatternEdge(SURNAME, surname);
    person.addPatternEdge(NAME, name);
    person.setAttributeMatchExpression(
            "#{" + Node.TYPE_ATTRIBUTE + "}=='" + FEMALE + "'");

    familyMember.setAction("!=");
    familyMember.addPatternEdge("!=", NAME, surname);
    familyMember.setAttributeMatchExpression(
            "#{" + Node.TYPE_ATTRIBUTE + "}=='" + FAMILYMEMBER + "'");

    family.addPatternEdge("!=", DAUGHTER, familyMember);

    familyRegister.addPatternEdge("+", FAMILIES, newFamily);
    newFamily.setAction("+");
    newFamily.addPatternEdge("+", NAME, name);
    newFamily.setPatternAttribute("+", Node.TYPE_ATTRIBUTE, FAMILY);
    newFamily.addPatternEdge("+", DAUGHTER, newFamilyMember);
    newFamilyMember.setAction("+");
    newFamilyMember.addPatternEdge("+", NAME, surname);
    newFamilyMember.setPatternAttribute("+", Node.TYPE_ATTRIBUTE, FAMILYMEMBER);
```

4 Results

Fastestestest solution ever.

5 Conclusion

As this is the first use of Yage in a context outside of examples implemented as proof-of-concept, we are very pleased by the results achieved utilizing this new tool. As pointed out in section 2 one of our biggest obstacles was transforming the given case structure to match our graph model. We were not able to port all cases to our tool due to the large amount of modifications applied during benchmarking and the work needed to reimplement them. Nevertheless the two cases solved with Yage are once the data was available a very straightforward process of modeling the given rules according to the case description. Performance suffers a little from the fact that we utilize one huge data structure containing both input and output data, i.e. the FamilyRegister as well as the PersonRegister.

References

[Alchemy.js] `http://graphalchemist.github.io/Alchemy/`

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