The Fulib Solution to the TTC 2019 Truth Table to Binary Decision Diagram Case

Albert Zündorf, Kassel University, zuendorf@uni-kassel.de

1 Introduction

I work on model transformations (based on graph transformations) for quite some time now. It started with the Progres system, [Zün94]. Then we developed the Fujaba environment [NNZ00]. To get rid of GUI problems we switched to SDMLib [SDMLib] in 2012. In September 2018, we switched to Fulib, the Fujaba library, [Fulib]. In [ZGR17], we propose to use tables to report the matches of model patterns (or graph transformation rules). This has been implemented in SDMLib and has been re-implemented in Fulib. Our (Fulib) tables provide us with a lot of functionality that you know from relational algebra, basically join, project, and select. In addition the Fulib tables provide functionality to add new columns e.g. for attribute values of matched objects.

The Truth Table to Binary Decision Diagram Case starts with loading a truth table. As the generic Fulib table is able to represent a truth table, easily, I programmed a simple reader that loads a given truth table model directly into a Fulib table. Then, I programmed a model transformation *splitTruthTable(tt,varName)* that creates two new tables, a true table and a false table. Basically, splitTruthTable selects all rows of the input table that have value true (or false) for the varName column and then it drops the varName column. Then I create a BDD node for the corresponding varName. Finally, I repeat these steps recursively on the two new tables. Currently, I stop the recursive call as soon as only one row is left in the table. This might easily be improved to stop already, when all rows have the same output values.

In Section 2, I will first outline how Fulib tables serve as model transformations. Then, Section 3 discusses the Fulib solution for generating BDDs. Finally, Section 4 shows our results.

2 Fulib Tables

Listing 1 shows some examples for Fulib tables. In line 1 we generate a BDD from a Truth Table. This will be discussed in Section 3. Line 2 creates an *ObjectTable* with one column and one row holding the *bdd* object. Line 3 expands this initial table with all ports reachable from the elements of the first column. As *bdd* has 4 input ports and 2 output ports, the result is a table with two columns and 6 rows where each row holds the *bdd* and one of the port objects. *expandLink* returns a new table object that refers to the same base table as the *rootTable* but further *expand* operations are applied to the port objects of the second column. Thus, the *expandString* operation of Line 4 visits all port objects of column two and reads their *name* attribute and adds the value to each row within a third column. The result is a *StringTable* that applies further operations to the elements of the third column of our table. Per default, the columns get names "A", "B", ... However, you may provide your own column name as initial parameter to an *expand* operation. Thus, the third column of our table has name "Ports". Now, Line 5 reduces our table to contain only the list of columns passed as *String...* parameter. Alternatively, we could call *dropColumns* and pass the names of the columns we want to get rid of. Line 6 prints the resulting table, the output is shown in Line 11 to 18 of Listing 1. On an *ObjectTable* you may call *toSet* in order to get the set of objects contained in the associated table column. On a *StringTable* you may call *toList* in order to get the list of strings (potentially with double values), cf. Line 7, Line 8, and Line 20.

ObjectTables also provide a *t1.hasLink(linkName,t2)* operation that allows to reduce the underlying table to all rows where the objects of the column associated with t1 and associated with t2 are connected via a *linkName*

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

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  BDD bdd = FulibSolution.doTransform("models/GeneratedI4O2Seed42.ttmodel");
2  ObjectTable rootTable = new ObjectTable(bdd);
3  StringTable nameTable = rootTable.expandLink("ports")
4                                   .expandString("Ports", "name");
5  rootTable.selectColumns("Ports");
6  System.out.println(rootTable);
7  ArrayList<String> nameList = nameTable.toList();
8  System.out.println(nameList);
9
10 /* Output:
11 | Ports |
12 |  ---  |
13 | I0    |
14 | I1    |
15 | I2    |
16 | I3    |
17 | O0    |
18 | O1    |
19
20 [I0, I1, I2, I3, O0, O1]
21 */
```

Listing 1: Fulib Table Model Queries

link. There is also a *filter(lambda)* operation that applies the passed lambda function to all rows and keeps only the rows where the lambda function computes true. Finally, there is an *addColumn(colName,lambda)* operation that runs the passed lambda function on each row and adds the result to the row. This may e.g. be used to create new neighbor objects or do some other model transformation.

Altogether, Fulib tables allow to build complex model queries and model transformations. These Fulib model transformations are still missing some common features provided by modern model transformation languages, but to our surprise, this small interface already suffices for most of the example cases that we have studied. Actually, for the Truth Table to Binary Decision Diagram case, we had to add some more features which is discussed below.

3 Generating BDDs

First, I decided not to load the *ttmodel* files as EMF object models but to read the data directly into a Fulib table. Thus, I just read the *ttmodel* file line by line and use a regular expression to match lines starting with "<ports" and to retrieve the contained port name. Then I extended the Fulib table class with an *addColumn(colName)* operation that I use to add a column for each retrieved port (no rows yet).

Internally, a Fulib Table consists of an *ArrayList<ArrayList<Object>>* and an additional *LinkedHashMap<String,Integer>* used to translate column names into *ArrayList* index values. Thus, while reading the *ttmodel* file line by line, for each "<rows" line, I create a new *ArrayList* and for each "<cells" line I add either 0 or 1 to this *ArrayList*. On a "</rows" line, I add the *ArrayList* to the current Fulib table. Yes, this very much relies of a very regular structure of the *ttmodel* files, but it works great for the generated example models.

Next, I programmed a *splitTable(tt,colName)* model transformation, cf. Listing 2. Basically, *splitTable* derives a *falseTable* and a *trueTable* from a given table *truthTable*. The new tables get all columns from the old table, without the *varName* column, cf. Lines 4 to 9. Then, we iterate through all rows of the input table (Line 11), clone it (Line 12), and remove the value that corresponds to the current *varName* (Line 13). The resulting reduced row is then added either to the *falseTable* or to the *trueTable* depending on the row value corresponding to the current *varName* (Lines 14 to 20).

Listing 3 shows my overall Truth Table to Binary Decision Diagram transformation *doOneBDDLevel*. In Line 17 *doOneBDDLevel* just chooses an (the first) input port and then splits the current table at this port (name) (Line 18). Lines 19 to 21 create the corresponding *SubTree* node. Then, Line 22 and 23 call *doOneBDDLevel* for

```
1  private ArrayList<FulibTable> splitTable(FulibTable truthTable, String varName) {
2     FulibTable falseTable = new FulibTable();
3     FulibTable trueTable = new FulibTable();
4     for (String key : truthTable.getColumnMap().keySet()) {
5        if (! key.equals(varName)) {
6           falseTable.addColumn(key);
7           trueTable.addColumn(key);
8        }
9     }
10    int varIndex = truthTable.getColumnMap().get(varName);
11    for (ArrayList row : truthTable.getTable()) {
12       ArrayList<Integer> clone = (ArrayList<Integer>) row.clone();
13       clone.remove(varIndex);
14       int value = (Integer) row.get(varIndex);
15       if (value == 1) {
16          trueTable.addRow(clone);
17       }
18       else {
19          falseTable.addRow(clone);
20       }
21    }
22    ArrayList<FulibTable> result = new ArrayList<>();
23    result.add(falseTable);
24    result.add(trueTable);
25    return result;
26 }
```

Listing 2: Split Table

the sub tables *falseTable* and *trueTable* recursively. Lines 24 and 25 add the resulting sub trees to the current sub tree. If the current *truthTable* has only a single row left (cf. Line 2), I generate a *Leaf* node with appropriate *Assignment*s (Lines 3 to 14).

Generally, the size of the generated BDD tree may be reduced by combining leaf nodes with similar output. Thus, we might improve our tree generation by checking whether all rows of a given table have the same output values. With a Fulib table, we may drop all input columns. The *dropColumns* operation would result in a table with only the output columns remaining and it would automatically eliminate all row duplicates. Thus we might then check for a single row or single vector of output values. However, such a *dropColumns* operation uses runtime that is linear to the number of rows and we are not sure that it pays off. Similarly, the size of the BDD tree may depend on the order of the input ports that you use for splitting. After all, within the benchmark the input examples are generated using random output values. Thus, there is little hope that a smart ordering of variables achieves an early pruning of the BDD tree. Thus, I did not investigate in a smart port selection algorithm but I just split the tables on the first column.

The reader may argue, that my *splitTable* operation is just plain Java and thus no real model transformation. On the other hand, our input is a Truth *Table* and thus I argue that a (Fulib) table is indeed an appropriate model for our input. Similarly, operations like *expandLink*, *addColumn*, and *addRow* are appropriate model transformations for a table model and these are the operations *splitTable* relies on.

4 Results

Figure 1 shows some runtime results for the Fulib solution in comparison with the default EMFSolutionATL. Figure 1 shows just the time used for the actual transformation in seconds. For 4 input ports and 2 output ports EMFSolutionATL needs about 0,45 seconds on my computer. For 8 input ports and 2 output ports EMFSolutionATL needed about 40 seconds on the same test. I have omitted this and larger measurements as you would no longer see the Fulib results. Fulib solves the case with 15 inputs and 5 outputs within 0.18 seconds. I have validated the results of the Fulib solution using the validator provided with the case study. It passes all

```
1  private static Tree doOneBDDLevel(BDD bdd, FulibTable truthTable) {
2      if (truthTable.getTable().size() == 1) {
3          Leaf leaf = BDDFactory.eINSTANCE.createLeaf();
4          for (String key : truthTable.getColumnMap().keySet()) {
5              if (key.startsWith("O")) {
6                  int value = (int) truthTable.getValue(key, 0);
7                  Assignment assignment = BDDFactory.eINSTANCE.createAssignment();
8                  OutputPort port = getOutputPort(bdd, key);
9                  assignment.setPort(port);
10                 assignment.setValue(value == 1);
11                 leaf.getAssignments().add(assignment);
12             }
13         }
14         return leaf;
15     }
16     else {
17         String varName = chooseVariable(truthTable);
18         ArrayList<FulibTable> kidTables = splitTable(truthTable, varName);
19         Subtree subtree = BDDFactory.eINSTANCE.createSubtree();
20         InputPort inputPort = getInputPort(bdd, varName);
21         subtree.setPort(inputPort);
22         Tree falseSubTree = doOneBDDLevel(bdd, kidTables.get(0));
23         Tree trueSubTree = doOneBDDLevel(bdd, kidTables.get(1));
24         subtree.setTreeForZero(falseSubTree);
25         subtree.setTreeForOne(trueSubTree);
26         return subtree;
27     }
28 }
```

Listing 3: Build the BDD

tests.

I am not sure, how the difference in runtime may be explained. First of all, an *ArrayList<ArrayList<Object>>* is a pretty compact and fast model. If we count each *ArrayList* as one object, the case with 15 + 5 ports has 2^{15} i.e. about 32 000 rows. Within the original EMF Truth Table Model, there are extra objects for each cell. These are additional 20 cell objects per row resulting in 640 000 cell objects. Thus, this may account for some factor of 20 in memory and accordingly in runtime usage. Still, it seems that the EMFSolutionATL is not in the same runtime complexity class as the Fulib solution. Maybe EMFSolutionATL does some more optimization or it does some more searching through the entire table during query matching. To be honest, the current Fulib solution works only for regular input (no don't cares, always the same order of cells). While I claim that this could be fixed within a runtime overhead of less than a factor of 2, this may also explain some of the differences.

Github: https://github.com/azuendorf/FulibSolutionTTC2019

Docker: zuendorf/fulib-solution-ttc2019

References

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. 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

[Fulib] FULib - The Fujaba Library *www.fulib.org/github* June, 2019.

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

[ZGR17] Zündorf A., Gebauer D., Reichmann C. (2017) Table Graphs. In: de Lara J., Plump D. (eds) Graph Transformation. ICGT 2017. Lecture Notes in Computer Science, vol 10373. Springer, Cham

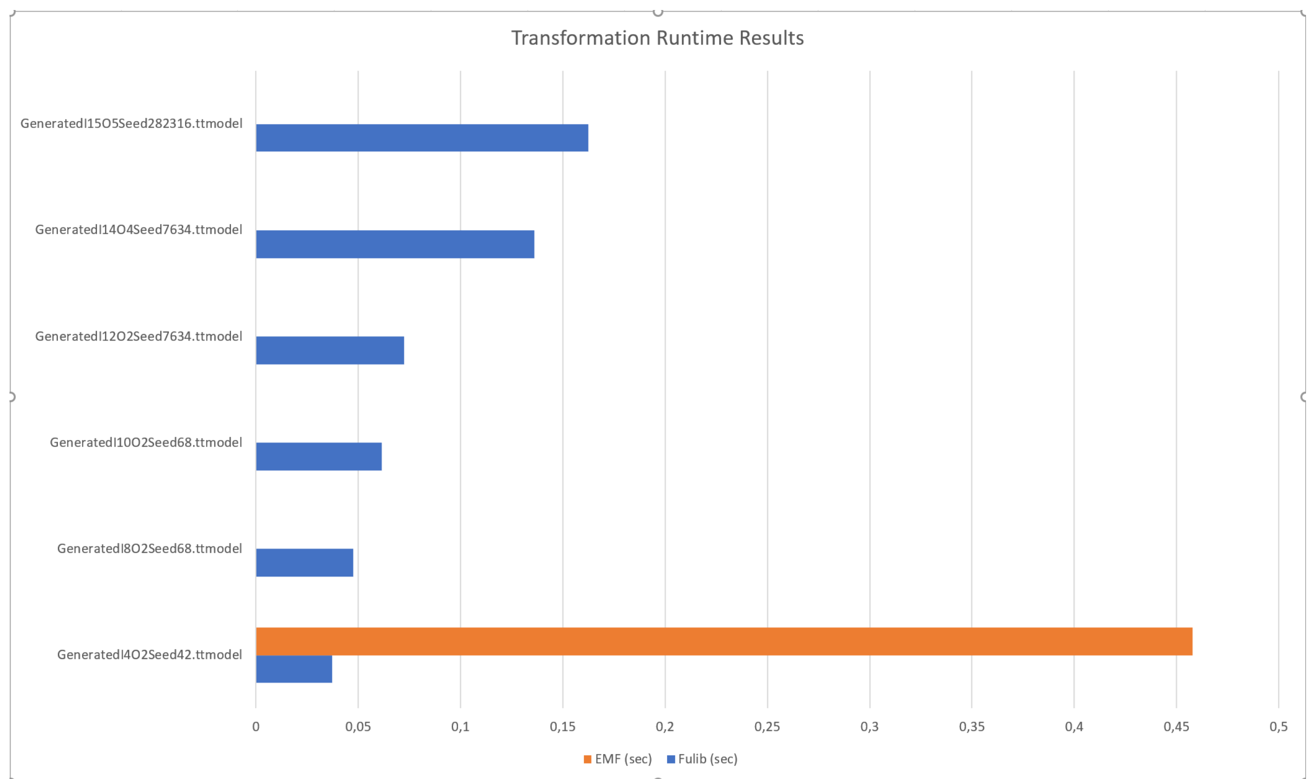

Figure 1: Runtime results

[Zün94] Zündorf, Albert. "Graph pattern matching in PROGRES." International Workshop on Graph Grammars and Their Application to Computer Science. Springer, Berlin, Heidelberg, 1994.