The TTC 2019 TT2BDD Case

Antonio García-Domínguez
Aston University
B4 7ET, Birmingham, United Kingdom
a.garcia-dominguez@aston.ac.uk

May 9, 2019

Model transformation tools have reached a considerable level of maturity in the core features, and are currently developing in many directions. Some tools are focusing on providing higher performance for large models or complex transformations. Others focus on bidirectionality, visualisation, traceability, or verifiability, among other research directions. Whereas past cases in TTC have focused on specific research directions, this case study presents a well-known simple transformation and welcomes researchers to apply their research to it. The aim of this case is to serve as a showcase of the various directions that model transformation research is going towards at the moment.

1 Introduction

Past editions of the Transformation Tool Contest have focused on a variety of topics:

- In 2018, the Quality-based Software Selection and Hardware-Mapping case discussed optimisation-oriented model transformations (with a combination of performance and solution quality). The Social Media Live Case considered performance in updating model views as models changed (with a strong preference for approaches supporting incrementality).
- In 2017, the Smart Grid case focused on incrementality, the Families to Persons case discussed bidirectional transformations, State Elimination focused on performance and the live case on Transformation Reuse discussed mechanisms to share complex logic across multiple versions of a transformation.
- In 2016, optimisation-oriented model transformations were discussed in considerable breadth through the Class Responsibility Assignment case, and an alternative dataflow-based notation for model transformation was evaluated in the live case study.

While these were notable examples of realistic transformations, they were narrowly focused on a specific topic,and their inherent complexity discouraged some attendees from trying their hand with their own research agenda on the transformation.

In this case, we propose a broader contest that welcomes all active lines of work on model transformation, based on a simpler, well-known transformation from the ATL Zoo[1]: the TT2BDD (Truth Tables to Binary Decision Diagrams) example transformation. Striving for raw performance is an option, but the case welcomes approaches that focus on other attributes of interest to a high-quality model transformation. This includes attributes such as verifiability, traceability, bidirectionality, or understandability, but solution providers are welcome to propose their own attributes of interest. In general, this case is proposed as a showcase of the current variety of model transformation tools.

The rest of the document is structured as follows: Section 2 describes the TT2BDD transformation. Section 3 suggests several tasks of interest that could be tackled in a solution (authors are free to propose their own tasks of interest). Section 4 mentions the benchmark framework for those solutions that focus on raw performance. Finally, Section 5 mentions an outline of the initial audience-based evaluation across all solutions, and the approach that will be followed to derive additional prizes depending on the attributes targeted by the solutions.

2 Transformation Description

This section introduces the Truth Tables to Binary Decision Diagram transformation, with a description of the input and output metamodels, and an outline of an implementation.

2.1 Input Metamodel: Truth Tables

The input metamodel is shown on Figure 1. A TruthTable object acts as the root of the model, and contains a collection of Ports and Rows. Ports come in two types: InputPorts and OutputPorts. Rows contain sequences of Cells, which assign values to the InputPorts and OutputPorts of the table.

Automated EMF opposite references are used liberally in the metamodel to simplify the specification of the transformation. *owner* is used to access the container from several child objects (e.g. the TruthTable of a Row), and it is possible to see which *cells* referenced a specific Port.

A sample model is shown on Table 1. The truth table has four InputPorts named A to D, and one OutputPort named S. The first Row contains only 3 Cells, specifying that if A and B are 0, then S should be 0.

2.2 Output Metamodel: Binary Decision Trees

The output metamodel is shown on Figure 2. A BDD object acts as the root of the model, and contains the root of the Tree and a collection of Ports. Similarly to the

[1]https://www.eclipse.org/atl/atlTransformations/

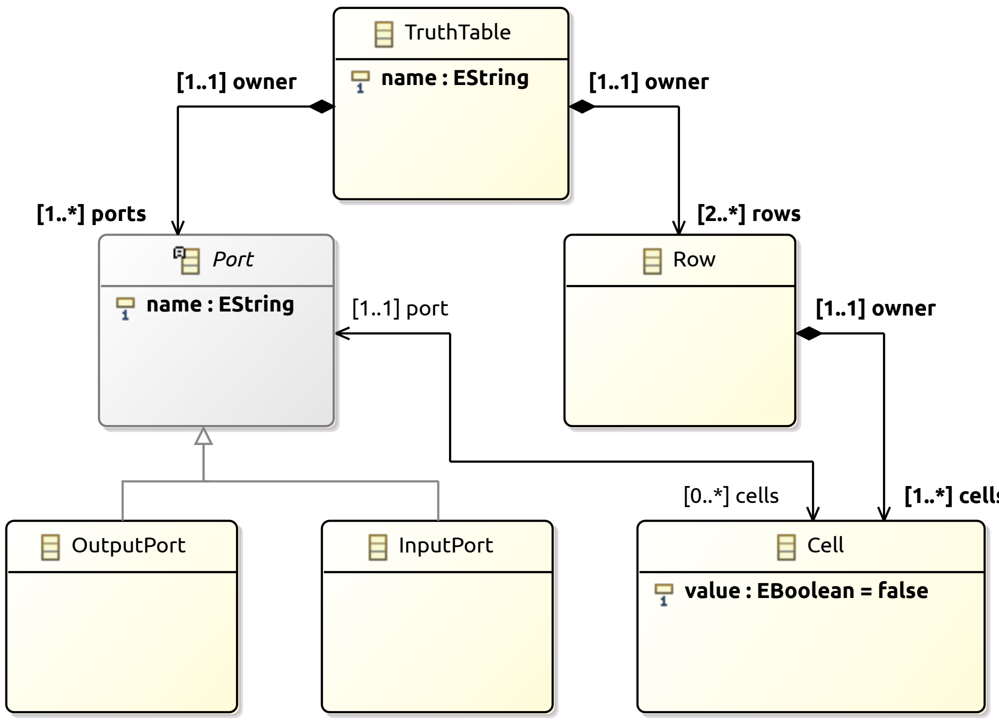

Figure 1: Class diagram for the input Truth Tables metamodel

A	B	C	D	S
0	0	-	-	0
0	1	0	0	1
0	1	0	0	1
0	1	0	1	0
0	1	1	-	0
1	0	0	0	0
1	0	1	0	1
1	-	-	1	0
1	1	0	0	1
1	1	1	0	0

Table 1: Example truth table: A to D are input ports and S is an output port. “-” means “ignored”.

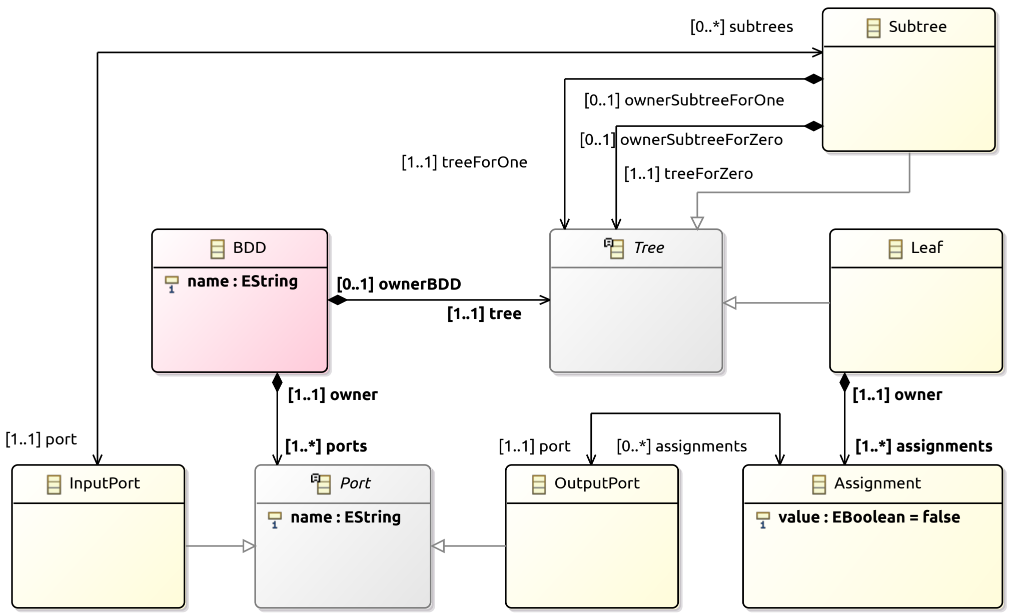

Figure 2: Class diagram for the output Binary Decision Diagram metamodel

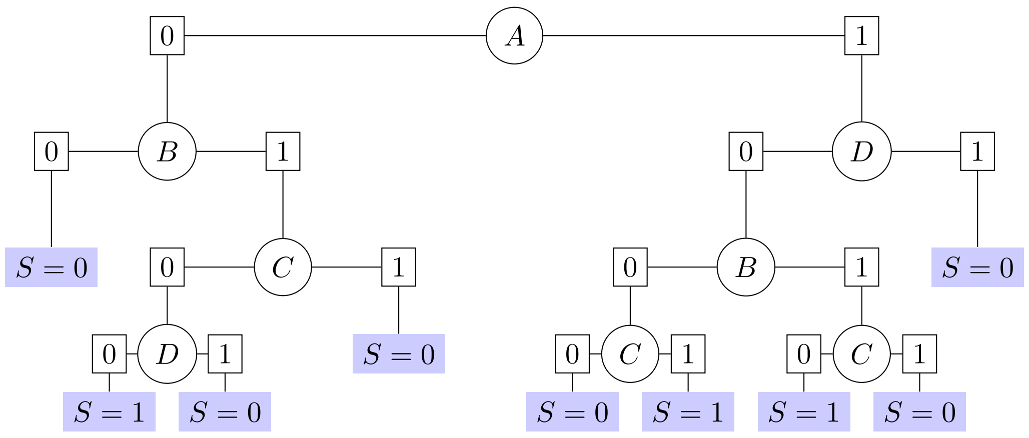

Figure 3: Equivalent BDD for the truth table on Table 1

Truth Tables metamodel, there are InputPorts and OutputPorts.

Tree is the common superclass for any node in the tree. Inner nodes check the value of an InputPort: if it is a false value, evaluation will proceed through the *treeForZero* Tree; otherwise, evaluation will go through the *treeForOne* Tree. Leaf nodes are Leaf objects, which provide an Assignment of a boolean value to each of the available OutputPorts.

The equivalent BDD for the truth table on Table 1 is shown on Figure 3. Subtrees are represented by the circle referencing an InputPort and their subtrees for when the port takes a 0 or 1 value. Assignments are represented by the shaded nodes that provide values to the OutputPorts.

2.3 Process Outline

The transformation essentially needs to construct a (preferably minimal) binary decision diagram that produces the same values for the output ports, given the same values in the input ports. Given the high interest about this problem in circuit design, many algorithms have been proposed in the literature.

Solution authors are welcome to implement a more optimal algorithm in their transformation tool. This section will only outline a simple approach that can be readily implemented without too much complication.

There are some basic mappings which are immediately obvious:

- Each TruthTable object should correspond to a BDD object, with the same name and equivalent Ports.
- Each InputPort and OutputPort should be mapped to an object of the BDD type with the same name, and should be assigned the same *name*.
- Each Row should become a Leaf node: the Cells for the OutputPorts will become Assignments.

The complexity is in deriving the inner nodes: the Subtree objects. One simple approach is to find a TT InputPort which is (ideally) defined in all the Rows, and turn it into an inner node (a Subtree) which points to the equivalent BDD InputPort and has two Trees:

- The zero subtree, produced from the Row(s) where the port was 0. This will be a Subtree if there are at least two rows in that situation: the transformation should proceed recursively in this case with those rows, excluding the input ports that have already been considered. If there is only one such row, this would simply point to the Leaf created above.
- The one subtree, produced in a parallel way to the zero subtree.

This simple approach does not necessarily ensure a minimal subtree, as in some points there may be multiple ports to choose from. It may require improvements for cases where there are no input ports which are defined in all available rows. Authors are welcome to try and implement a more efficient or optimal approach in their solutions.

3 Main Task

The obvious main task in this case is to implement the TT2BDD transformation in your favourite model transformation language, ensuring that the resulting BDDs are equivalent to the original truth table. To simplify the work involved, the case includes an Eclipse Modelling Framework implementation of the metamodels in the `ttc2019.metamodels` project within the `metamodels` folder, and a set of sample XMI input models conforming to the metamodels in the `models` folder.

The `models` folder includes two additional tools, which can be run with `java -jar tool.jar`:

- `generator.jar` can produce truth tables of an arbitrary number of input and output ports, given seed. Solution implementers may want to be careful with the values given, as truth tables grow exponentially as more input ports are added.
- `validator.jar` can validate if a BDD model is equivalent to a source truth table model, by evaluating all the rows in the TT model through the BDD and comparing the values of the OutputPorts. This tool is required to check that your transformation is correct.

Solutions are free to focus on any specific quality attribute of the transformation beyond optimality and performance. For instance, it may be useful to be able to prove that the transformation does indeed produce a BDD which is equivalent to the TT (even if suboptimal). It may also be interesting to be able to visualize the BDD being built from the TT, or to be able to trace which subtree came from which collection of rows.

The conciseness of the transformation may also be of interest, given that its recursive nature may be difficult to map to some existing transformation languages.

4 Benchmark Framework

If focusing on performance, the solution authors should integrate their solution with the provided benchmark framework. It is based on that of the TTC 2017 Smart Grid case[2], and supports the automated build and execution of solutions. For this specific case study, the visualisation of the results is currently disabled.

The benchmark consists of three phases:

1. **Initialization**, which involves setting up the basic infrastructure (e.g. loading metamodels). These measurements are optional.
2. **Load**, which loads the input models.
3. **Run**, which runs the transformation, produces the BDD and saves the file.

[2] `https://www.transformation-tool-contest.eu/2017/solutions_smartGrid.html`

4.1 Solution requirements

Each solution wishing to use the benchmarking framework should print to the standard output a line with the following fields, separated by semicolons (";"):

- **Tool**: name of the tool.
- **Model**: name of the model being transformed.
- **RunIndex**: the index of the repetition of the transformation.
- **Phase**: the name of the phase.
- **MetricName**: the name of the metric. It may be the used **Memory** in bytes, or the wall clock **Time** spent in integer nanoseconds.

To enable automatic execution by the benchmark framework, solutions should add a subdirectory to the `solutions` folder of the benchmark with a `solution.ini`q file stating how the solution should be built and how it should be run. Solution authors will want to study the available `solution.ini` for the sample ATL solution, and adjust its settings in order to run the appropriate commands for building and running their solutions.

The repetition of executions as defined in the benchmark configuration is done by the benchmark. For 5 runs, the specified command will be called 5 times, passing any required information (e.g. run index, or input model name) through environment variables. Solutions must not save intermediate data between different runs: each run should be entirely independent.

The name and absolute path of the input model, the run index and the name of the tool are passed using environment variables `Model`, `ModelPath`, `RunIndex` and `Tool`. Solution authors are suggested to study the example ATL solution on how to use these values to run their transformation.

4.2 Running the benchmark

The benchmark framework only requires Python 2.7 to be installed. Furthermore, the solutions may imply additional frameworks. We would ask solution authors to explicitly note dependencies to additional frameworks necessary to run their solutions.

If all prerequisites are fulfilled, the benchmark can be run using Python with the command `python scripts/run.py`. Additional options can be queried using the option `--help`. The benchmark framework can be configured through the `config/config.json` file: this includes the input models to be evaluated (some of which have been excluded by default due to their high cost with the sample solution), the names of the tools to be run, the number of runs per tool+model, and the timeout for each command in milliseconds.

5 Evaluation

Given the call for a broader set of research interests in this transformation, the evaluation will operate on two dimensions:

- Is it a high-quality model transformation? The transformation should be complete, correct, easy to understand, efficient, and produce optimal results.

  The `validator.jar` will be used to check the produced BDDs against the source truth tables, and the authors will need to provide a convincing argument about the correctness of the solution.

  Attendees to the contest will evaluate the understandability of the solution, and the benchmark framework will provide independent measurements of memory and time usage.

  Tree sizes for the BDDs will be considered for the optimality of the transformations: the smaller, the better.

- Does it highlight a promising research direction? Although the transformation may not be entirely complete or may be harder to understand, it may serve as an example of an active research area within model transformations that the community may wish to showcase.

  This may include aspects such as incrementality, bidirectionality, traceability, verifiability, or the ability to visualize interactively the transformation, among other areas of interest in the field.

  Depending on the submitted solutions, the awards for this dimension may change. If there are no common areas of interest in the solutions, an audience-driven "Most Promising" award will be awarded. If there are common themes between solutions, dedicated 1st/2nd/3rd place awards for those research areas may be awarded.