Proceedings of the 12th International Workshop
Transformation Tool Contest

TTC 2019

Antonio Garcia-Dominguez
Georg Hinkel
Filip Křikava

Preface

The aim of the Transformation Tool Contest (TTC) series is to compare the expressiveness, the usability, and the performance of transformation tools along a number of selected case studies. A deeper understanding of the relative merits of different tool features will help to further improve transformation tools and to indicate open problems.

This contest was the tenth of its kind. For the fifth time, the contest was part of the Software Technologies: Applications and Foundations (STAF) federation of conferences. Teams from the major international players in transformation tool development have participated in an online setting as well as in a face-to-face workshop.

In order to facilitate the comparison of transformation tools, our programme committee selected three challenging cases via single blind reviews for which there were together ten solutions.

These proceedings comprise descriptions of the case study and all of the accepted solutions. In addition to the solution descriptions contained in these proceedings, the implementation of each solution (tool, project files, documentation) is made available in public version control repositories. Some solution are also available via the SHARE platform (http://share20.eu).

TTC 2019 involved open (i.e., non anonymous) peer reviews in a first round. The purpose of this round of reviewing was that the participants gained as much insight into the competitors' solutions as possible and also to identify potential problems. At the workshop, the solutions were presented. The expert audience judged the solutions along a number of case-specific categories, and prizes were awarded to the highest scoring solutions in each category. Finally, the solutions appearing in these proceedings were selected by our programme committee via single blind reviews. The full results of the contest are published on our website[1].

Besides the presentations of the submitted solutions, the workshop also comprised a live contest. That contest involved creating a solution for transformation reuse in the presence of multiple inheritance and redefinitions. The live contest was announced to all STAF attendees and participants were given four days to design, implement and test their solutions. The contest organisers thank all authors for submitting cases and solutions, the contest participants, the STAF local organisation team, the STAF general chair Gabriele Taentzer, and the program committee for their support.

19th July, 2019
Eindhoven, The Netherland

Antonio Garcia-Dominguez
Georg Hinkel
Filip Křikava

[1] `http://www.transformation-tool-contest.eu/`

Organisation

The Transformation Tool Contest has been organized by TUE in Eindhoven, The Netherland.

Program Committee

Konstantinos Barmpis	University of York, United Kingdom
Juan Boubeta-Puig	University of Cádiz, Spain
Erwan Bousse	Vienna University of Technology, Austria
Gwendal Daniel	AtlanMod - Inria, France
Antonio Garcia-Dominguez	Aston University, United Kingdom
Georg Hinkel	FZI Research Center of Information Technology, Germany
Tassilo Horn	SHD, Germany
Akos Horvath	Budapest University of Technology and Economics, Hungary
Filip Křikava	Czech Technical University, Czech Republic
Arend Rensink	University of Twente, The Netherlands
Massimo Tisi	Ecole des Mines de Nantes, France
Gergely Varro	Technische Universitat Darmstadt, Germany
Ran Wei	University of York, United Kingdom

Table of Contents

Truth Tables to Binary Decision Diagrams

Truth Tables to Binary Decision Diagrams . . . 3
Antonio Garcia-Dominguez and Georg Hinkel
An NMF solution to the TTC 2019 Truth Tables to Binary Decision Diagrams Case . . . 9
Georg Hinkel
The Fulib Solution to the TTC 2019 Truth Table to Binary Decision Diagram Case . . . 15
Albert Zuendorf
YAMTL Solution to the TTC 2019 TT2BDD Case . . . 21
Artur Boronat
Transforming Truth Tables to Binary Decision Diagrams using Relational Reference Attribute Grammars . . . 27
Johannes Mey, René Schöne, Christopher Werner and Uwe Assmann
Applying formal reasoning to model transformation - The Meeduse solution - . . . 33
Akram Idani, German Vega and Michael Leuschel
Transforming Truth Tables to Binary Decision Diagrams using the Role-based Synchronization Approach . . . 45
Christopher Werner, Rico Bergmann, Johannes Mey, René Schöne and Uwe Assmann
Truth Tables to Binary Decision Diagrams in Modern ATL . . . 51
Dennis Wagelaar, Théo Le Calvar and Frédéric Jouault

BibtexXML to Docbook Consistency Case Live Case

The TTC 2019 Live Case: BibTeX to DocBook . . . 61
Antonio Garcia-Dominguez and Georg Hinkel
An NMF solution to the TTC 2019 Live Case . . . 67
Georg Hinkel
YAMTL Solution to the TTC 2019 BibtexToDocBook Case . . . 73
Artur Boronat

Part I.

Truth Tables to Binary Decision Diagrams

Truth Tables to Binary Decision Diagrams

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

Georg Hinkel
Tecan Software Competence Center
55252, Mainz-Kastel, Germany
georg.hinkel@tecan.com

Abstract

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 [4] discussed optimisation-oriented model transformations (with a combination of performance and solution quality). The Social Media Live Case [7] 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 [6], the Families to Persons case discussed bidirectional transformations [1], State Elimination focused on performance and the live case on Transformation Reuse [5] 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 [2], and an alternative dataflow-based notation for model transformation was evaluated in the live case study [3].

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

This year, we proposed a broader contest that welcomed all active lines of work on model transformation. It was based on a simpler, well-known transformation from the ATL Zoo [8]: TT2BDD (Truth Tables to Binary Decision Diagrams). Striving for raw performance was an option, but the case welcomed approaches that focused on other attributes of interest of a high-quality model transformation: for example, verifiability, traceability, bidirectionality, or understandability. In general, this case was proposed as a showcase of the current variety of model transformation tools. All resources for this case are available on Github[1].

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

[1] https://github.com/TransformationToolContest/ttc2019-tt2bdd

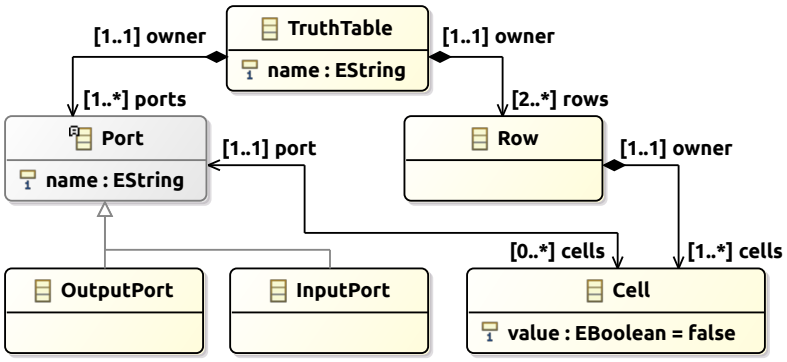


Figure 1: Class diagram for the input Truth Tables metamodel

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

A	B	C	D	S
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".

The rest of the document is structured as follows: Section 2 describes the TT2BDD transformation. Section 3 several tasks of interest that should 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. The TruthTable class is 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 original output metamodel from the ATL Zoo version is shown on Figure 2. The BDD class is the root of the model, and contains the root of the Tree and a collection of Ports. Similarly to the Truth Tables metamodel, there are InputPorts and OutputPorts.

Figure 3 shows the changes introduced into a revised version of the original metamodel, which allow the BDD to be a rooted acyclic graph and not just a tree. This would allow the BDD to reuse common subtrees, which would produce more compact circuits in some situations. The case provides both versions as separate metamodels: solution implementers were free to choose either version.

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

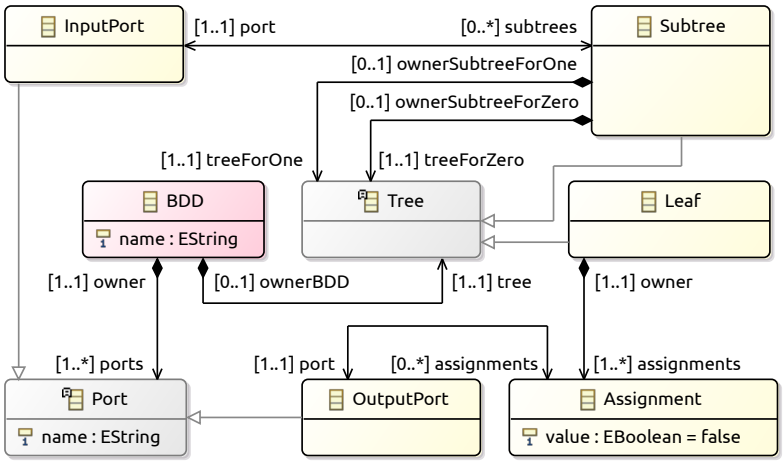


Figure 2: Class diagram for the output Binary Decision Diagram metamodel, in the original tree-based version.

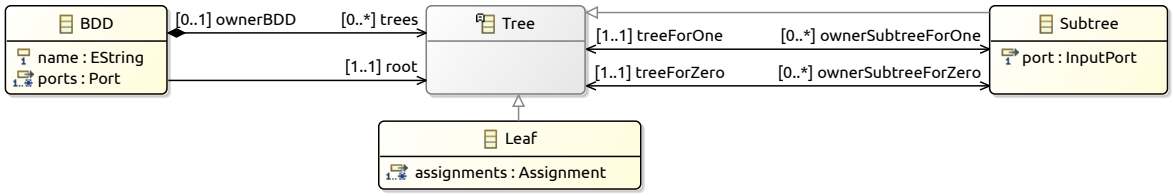


Figure 3: Changed classes in the graph-based variant of the Binary Decision Diagram metamodel (2019-05-26). Omitted clases remain the same.

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 4. 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 highlighted 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.
- 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.

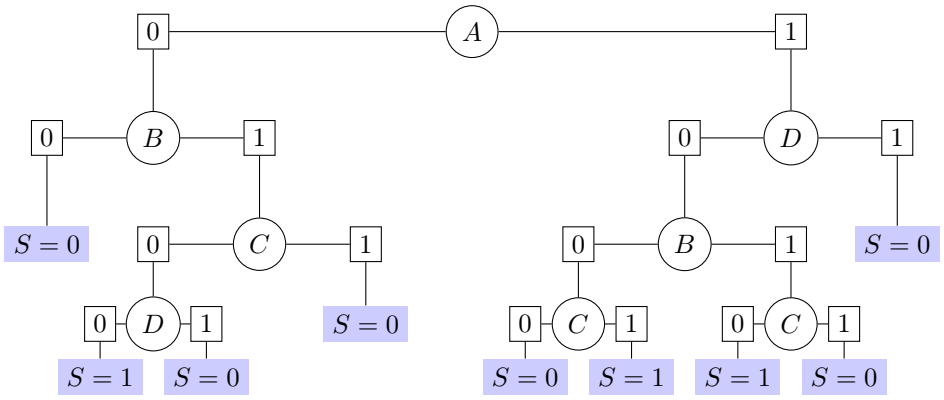


Figure 4: Equivalent BDD for the truth table on Table 1.

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.

3 Main Task

The main task was implementing the transformation, ensuring that the BDDs were equivalent to the original truth table. To simplify the work involved, the case included an Eclipse Modelling Framework implementation, and a set of sample XMI input models conforming to the metamodels.

The `models` folder includes two additional tools:

- A generator that produced truth tables of an arbitrary number of input and output ports, given a seed.
- A validator that checked if a BDD model was equivalent to a source truth table model, by evaluating all possible input combinations through the BDD and comparing the values of the OutputPorts.

Solutions could 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). One of the solutions did focus on verifiability.

4 Benchmark Framework

If focusing on performance, the solution authors had to integrate their solution with the provided benchmark framework. It is based on that of the TTC 2017 Smart Grid case [6], and supports the automated build and execution of solutions. The framework consists of a Python 3.3 script which directs the process, and a set of R scripts which perform basic data analysis and reporting.

The benchmark consisted of three phases: **Initialisation** (setting up the basic infrastructure), **Load** (for loading the models), and **Run** (for executing the transformation and saving the results).

Solutions had to be forks of the main Github project[2], for their later integration into the main Github repository. The solutions could be implemented in any language: they only had to print to the standard output lines in a specific format, mentioning the memory usage and wall clock time spent in transforming each model.

The solutions had to include a `solution.ini` file with instructions on how to automatically build, test, and run them. Solutions were run multiple times, as indicated in the main configuration of the benchmark. To ensure reproducibility, the solutions were integrated into a Docker image, which is automatically built by Docker Hub on each push[3].

5 Evaluation

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

[2] https://github.com/TransformationToolContest/ttc2019-tt2bdd
[3] https://hub.docker.com/r/bluezio/ttc2019-tt2bdd-git

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

  The validator was used to check the produced BDDs against the source truth tables, and the authors had to provide a convincing argument about the correctness of the solution.

  The understandability of the solution was evaluated through an audience survey, and the Docker image was used by the contest organizers to provide independent measurements of memory and time usage in identical conditions. Particularly, Google Cloud Compute `c2-standard-4` images were used.

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

- Did 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.

  As mentioned before, there was one solution which focused on the verifiability of the solution. This solution received an award for the promising results towards integrating verifiability in a model transformation language.

References

[1] Anthony Anjorin, Thomas Buchmann, and Bernhard Westfechtel. The Families to Persons Case. In *Proceedings of the 10th Transformation Tool Contest*, volume 2026, pages 27–34, Marburg, Germany, July 2017. CEUR-WS.org.

[2] Martin Fleck and Javier Troya. The Class Responsibility Assignment Case. In *Proceedings of the 9th Transformation Tool Contest*, volume 1758, pages 1–8, Vienna, Austria, July 2016. CEUR-WS.org.

[3] Antonio García-Domínguez. TTC'16 live contest case study: execution of dataflow-based model transformations. https://www.transformation-tool-contest.eu/2016/livecontest.html, July 2016. Last accessed on 2019-05-10. Archived on http://archive.is/gHEys.

[4] Sebastian Götz, Johannes Mey, René Schöne, and Uwe Aßmann. Quality-based Software-Selection and Hardware-Mapping as Model Transformation Problem. In *Proceedings of the 11th Transformation Tool Contest*, volume 2310, pages 3–11, Toulouse, France, June 2018. CEUR-WS.org.

[5] Georg Hinkel. The TTC 2017 live contest on transformation reuse in the presence of multiple inheritance and redefinitions. https://www.transformation-tool-contest.eu/2017/solutions_livecontest.html, July 2017. Last accessed on 2019-05-10. Archived on http://archive.is/gHEys.

[6] Georg Hinkel. The TTC 2017 Outage System Case for Incremental Model Views. In *Proceedings of the 10th Transformation Tool Contest*, volume 2026, pages 3–12, Marburg, Germany, July 2017. CEUR-WS.org.

[7] Georg Hinkel. The TTC 2018 Social Media Case. In *Proceedings of the 11th Transformation Tool Contest*, volume 2310, pages 39–43, Toulouse, France, June 2018. CEUR-WS.org.

[8] Guillaume Savaton. Truth Tables to Binary Decision Diagrams, ATL Transformations. https://www.eclipse.org/atl/atlTransformations/#TT2BDD, February 2006. Last accessed on 2019-05-14. Archived on http://archive.is/HdoHM.

An NMF solution to the TTC 2019 Truth Tables to Binary Decision Diagrams Case

Georg Hinkel

Am Rathaus 4b, 65207 Wiesbaden, Germany
georg.hinkel@gmail.com

Abstract

This paper presents a solution to the Truth Tables to Binary Decision Diagrams (TT2BDD) case at the Transformation Tool Contest (TTC) 2019. We demonstrate how the implicit incrementalization abilities of NMF can be used to automatically obtain an incremental algorithm for the presented case.

1 Introduction

In the past, the need for model transformation languages has often been argued with their improved conciseness and readability over general-purpose programming languages (GPLs). Recently, however, GPLs have become more declarative. As an example, pattern matching, an often used language feature, is going to be integrated in mainstream GPLs such as C#. As a consequence, programming languages tailored to a specific application area – such as model transformation languages – must prove that they give benefits to the developer that go beyond just a more concise specification, especially because conciseness does not necessarily imply understandability or maintainability, at least not throughout language boundaries. After all, using a dedicated model transformation languages usually comes with a wide range of disadvantages such a (compared to a mainstream GPL) much smaller community, smaller availability of trained staff, often worse editor support and worse modularity mechanisms.

The Truth Tables to Binary Decision Diagrams (TT2BDD) case at the TTC 2019 [1] aims to collect the state of the art in model transformations with regard to what these advantages are in the realm of model transformation languages and to apply them at a common scenario of converting the specification of a boolean function between two different formats.

Important examples of such advantages are incrementality and bidirectionality. Here, an incremental model transformation is a transformation that automatically adapts to changes of its input models. Such a model transformation is tedious to develop and it is also error-prone as it is easy to forget cases in which the output model of a transformation needs to be adapted. For model transformations, the trace between input and output model elements can be used to make an incremental model transformation efficient when new elements are added to the input model. In a similar fashion, the trace is helpful to invert a model transformations to yield a bidirectional model transformation. However, it is not obvious how the trace should be used to reach incrementality and bidirectionality, hence model transformation languages that encode this know how in a language and model transformation engine do provide a significant benefit over a batch specification of a model transformation language in a GPL.

One of the model transformation languages that can derive an incremental and/or bidirectional execution from a batch model transformation specification is *NMF Synchronizations* [2], [3], an extensible model transformation and synchronization language and system part of the .NET Modeling Framework (*NMF*, [4]). This paper

In: A. Garcia-Dominguez, G. Hinkel and F. Krikava (eds.): Proceedings of the 12th Transformation Tool Contest, Eindhoven, The Netherlands, 19-07-2019, published at http://ceur-ws.org

introduces the solution to the TT2BDD case using NMF, in particular NMF Synchronizations. The code for the solution is available online[1].

2 Synchronization Blocks and NMF Synchronizations

Synchronization blocks are a formal tool to run model transformations in an incremental (and bidirectional) way [2]. They combine a slightly modified notion of lenses [5] with incrementalization systems. Model properties and methods are considered morphisms between objects of a category that are set-theoretic products of a type (a set of instances) and a global state space Ω.

A (single-valued) synchronization block $\boldsymbol{S}$ is an octuple $(A, B, C, D, \Phi_{A-C}, \Phi_{B-D}, f, g)$ that declares a synchronization action given a pair $(a, c) \in \Phi_{A-C} : A \cong C$ of corresponding elements in a base isomorphism Φ_{A-C}. For each such a tuple in states (ω_L, ω_R), the synchronization block specifies that the elements $(f(a, \omega_L), g \nearrow (b, \omega_R)) \in B \times D$ gained by the lenses f and g are isomorphic with regard to Φ_{B-D}.

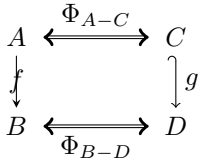


Figure 1: Schematic overview of unidirectional synchronization blocks

A schematic overview of a synchronization block is depicted in Figure 1. The usage of lenses allows such declarations to be enforced automatically and in both directions. The engine simply computes the value that the right selector should have and enforces it using the PUT operation. Similarly, a multi-valued synchronization block is a synchronization block where the lenses f and g are typed with collections of B and D, for example $f : A \hookrightarrow B*$ and $g : C \hookrightarrow D*$ where stars denote Kleene closures.

Synchronization Blocks have been implemented in NMF Synchronizations, an internal DSL hosted by C# [2]. For the incrementalization, it uses the extensible incrementalization system NMF Expressions [6]. This DSL is able to lift the specification of a model transformation/synchronization in three quite orthogonal dimensions:

- **Direction:** A client may choose between transformation from left to right or right to left
- **Change Propagation:** A client may choose whether changes to the input model should be propagated to the output model, also vice versa or not at all
- **Synchronization:** A client may execute the transformation in synchronization mode between a left and a right model. In that case, the engine finds differences between the model and handles them according to the given strategy (only add missing elements to either side, also delete superfluous elements on the other or full duplex synchronization)

This flexibility makes it possible to reuse the specification of a transformation in a broad range of different use cases. Furthermore, the fact that NMF Synchronizations is an internal language means that a wide range of advantages from mainstream languages, most notably modularity and tool support, can be inherited [7].

3 Solution

When creating a model transformation with NMF Synchronizations, one has to find correspondences between input and target model elements and how they relate to each other. The first correspondence is usually clear and is the entry point for the synchronization process afterwards: The root of the input model is known to correspond to the root of the output model, in our case the `TruthTable` element should correspond to the `BDD` element. Further, their names should match. There is also a correspondence between the ports of a truth table and the ports of a binary decision diagram and the ports used in the truth table have to be equivalent to the ports in the binary decision diagram.

In the formal language of synchronization blocks, these synchronization rules look like in Figure 2. Because the required model elements are directly part of the model, they are rather trivial to implement: The developer

[1] `https://github.com/georghinkel/ttc2019-tt2bdd`

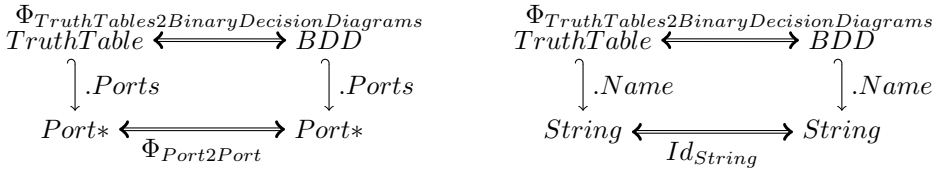


Figure 2: Synchronization block to synchronize ports and names

just needs to specify the properties that should be synchronized. If an isomorphism other than the identity should be used, it has to be specified as well.

```
1 public class TT2BDD : SynchronizationRule<TruthTable, BDD> {
2     public override void DeclareSynchronization() {
3         Synchronize(tt => tt.Name, bdd => bdd.Name);
4         SynchronizeMany(SyncRule<Port2Port>(), tt => tt.Ports, bdd => bdd.Ports);
5     }}
```

Listing 1: Definition of synchronization blocks from Figure 2 in NMF Synchronizations

The implementation for the synchronization blocks from Figure 2 is depicted in Listing 1. In particular, line 1 defines the isomorphism ϕ_{TT2BDD}, line 3 implements the right synchronization block from Figure 2 and line 4 implements the left one.

More interesting from an incrementalization and also bidirectionalization point of view is the synchronization between the rows of a truth table with the leafs of a binary decision diagram. Unlike the ports, the `Leaf` elements are spread over the entire output model as descendants of the decision diagram. In order to synchronize these with the input model, we need to create a virtual collection of all `Leaf` elements of a decision diagram. This virtual collection allows to specify the synchronization block to synchronize rows of the truth table with the leafs of the binary decision diagram as in Listing 2.

```
1 SynchronizeMany(SyncRule<Row2Leaf>(), tt => tt.Rows, bdd => new BDDLeafCollection(bdd));
```

Listing 2: Synchronizing the rows of the truth table with the leafs of the binary decision diagram

Virtual collections extend a model query with modification operators in case NMF is not able to add or remove elements from the underlying collection. The developer only has to implement a modification of the collection contents, in case of our solution adding an assignment to the input assignments of a leaf and adding a leaf to a binary decision diagram. Because NMF Synchronizations executes synchronization blocks from the more specific to the more general, it will first add input assignments to a leaf before adding that leaf to a binary decision diagram.

```
1 internal class BDDLeafCollection : CustomCollection<ILeaf> {
2   private readonly BDD _bdd;
3   public BDDLeafCollection(BDD bdd) : base(bdd.Descendants().OfType<ILeaf>()) { _bdd = bdd; }
4   public override void Add(ILeaf item) { ... }
5   public override void Clear() { _bdd.Tree = null; }
6   public override bool Remove(ILeaf item) { item.Delete(); return true; }
7 }
```

Listing 3: Virtual collection of the leafs of a binary decision diagram

The implementation of the virtual collection of leafs in a binary decision diagram is sketched in Listing 3. In line 3, a query expression of the collection is provided to the base class. NMF Expressions will use this query expression to receive notifications when the contents of the virtual collection change.

To add an assignment to a leaf, the implementation creates a new `SubTree` element with an according port and puts the leaf or its ancestor `SubTree` element as child for one or child for zero, as appropriate.

```
1 internal class TreeAssignmentsCollection : CustomCollection<TreeAssignment> {
2   private ILeaf _leaf;
3   public TreeAssignmentsCollection(ILeaf leaf) : base(
4     leaf.AncestorTree()
5         .Select(tree => new TreeAssignment((tree.Parent as ISubtree).Port, (tree.Parent as ISubtree).TreeForOne == tree.Child)))
6   { _leaf = leaf; }
7   ...
8 }
```

Listing 4: Virtual collection of input port assignments of a leaf from a binary decision diagram

For this, another virtual collection of assignments for a given leaf is created. An implementation is sketched in Listing 4. Again, obtainig the assignments for a given leaf can be easily specified as a query which NMF is able to automatically incrementalize. The developer only has to implement methods to add or remove an assignment.

The by far most complex bit of the implementation in our solution, beating the entire synchronization declaration in terms of lines of code, is the implementation to add a leaf to a binary decision diagram. Whereas the synchronization blocks allow a very declarative specification, this part of the implementation is very imperative and longer than the entire synchronization in terms of lines of code. This is necessary, because the declarative approach of NMF Synchronizations requires a clear semantics, whereas for the mapping of assignments has open conceptual problems: Meanwhile it is easily possible in the truth table model to insert conflicting information (there could be two rows having exactly the same input port cells but different output port cells), this is just not possible in the binary decision diagrams model.

In a first step, we collect the assignments for a leaf to ports. In case the decision diagram does not have any tree yet, we simply set that tree and return. Otherwise, we collect the assignments and keep a stack of inner tree nodes. Next, we go from the root of the binary decision diagram towards the leafs and select the path that should be taken for the leaf in question. That is, we traverse the tree, taking the collected assignments as a basis whether to walk the *TreeForOne* or *TreeForZero* reference until we find a spot where an equivalent inner node does not exist. There, we copy the `SubTree` elements for ports not yet processed along the path from the root.

4 Discussion

The solution shows how easy it is to incrementalize or bidirectionalize a model transformation if the representation of the information contained in the models is similar, such as in case of the input and output ports that have exactly the same structure. Here, the specification of the model transformation is little more than a mapping between types and members of input and output model.

To support bidirectional execution, our solution contains a third virtual collection, namely the collection of assignments of a row. However, here the implementation to add or remove such an assignment is rather easy, because it just consists of adding or deleting an appropriate cell. This is because the data structure chosen to synchronize the model contents essentially is the structure used by the truth tables model. Apart from that, the model transformation can be inverted automatically. Hence, bidirectional execution is a low hanging fruit for this model transformation.

The price for the incrementality is slightly more subtle. In principle, the incrementalization system of NMF is powerful enough to automatically incrementalize the entire model transformation without the developer having to implement a change propagation explicitly. However, the drawback here is that the few model modification implementations, adding or removing elements from one of the virtual collections, are executed in many more scenarios. For example, the virtual collection of assignments for a leaf currently does not consider the case where the leaf is actually contained in a binary decision diagram – a case that is not relevant for a batch transformation and non-trivial to implement.

Furthermore, the synchronization blocks force the transformation to process the rows one after another and independently for each property that is synchronized in a synchronization block. From a performance perspective, this is highly inefficient as it makes any form of preprocessing hard in cases where multiple data is woven together such as in the tree structure of a binary decision diagram. In particular, the NMF solution creates a range of `SubTree` elements just to carry over the information of assignments which then has to be parsed. As a result, the performance of the NMF solution is not as good as solutions that create the binary decision diagram all together.

Figure 3 shows the performance results of the solutions submitted to the TTC 2019. These results were produced through a Google Cloud Compute Engine c1-standard-4 machine with 16GiB of RAM and 30GB SSDs, using a Docker image produced through the Dockerfile in the root of the benchmark repository. It shows that the NMF solution is faster than the reference solution by multiple orders of magnitude for the medium-sized models, but then again significantly slower than the solutions using Fulib, reference attribute grammars or YAMTL. In particular, the slope of the graph on a logarithmic scale indicates that the NMF solution has a worse asymptotic complexity.

Last but not least, the fact that NMF automatically enforces bidirectional references and ensures referential integrity has a nitpick: Removing a model element from its container deletes this model element in NMF, which in turn causes all model elements referencing this model element to delete this reference (in order to avoid a reference to a deleted element). Hence, the *Port* reference of an inner node is reset once it is deleted, because it

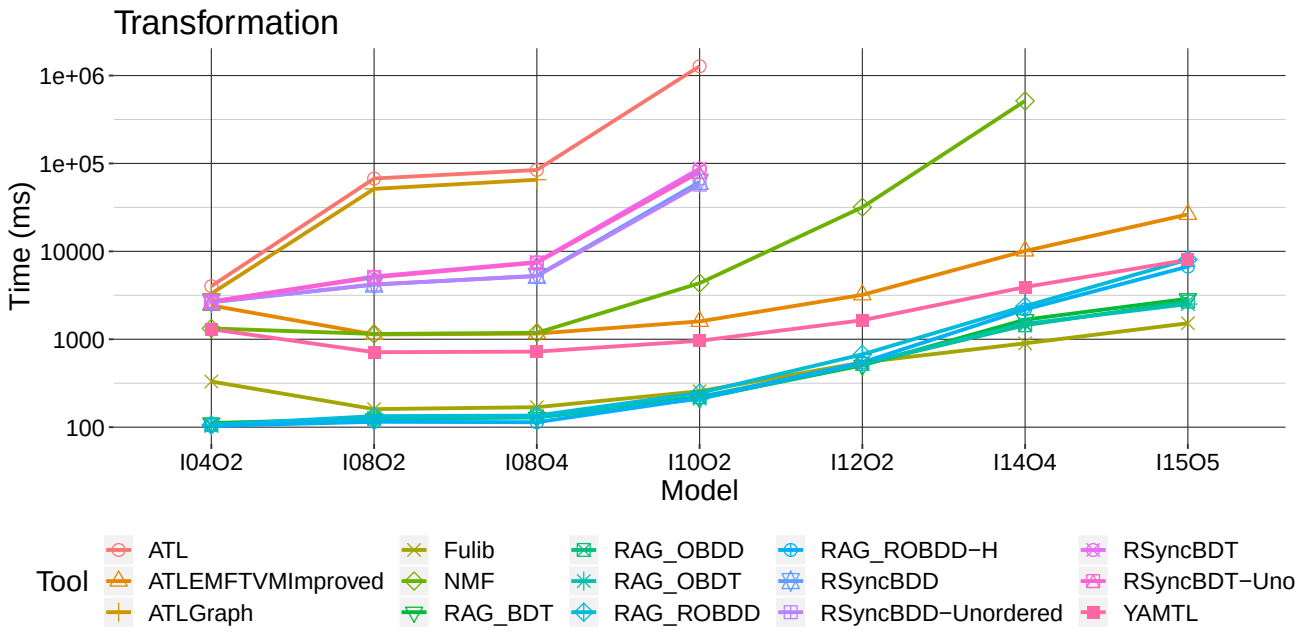


Figure 3: Performance results of the solutions at the TTC 2019

is an opposite direction reference of the port referencing its inner nodes (which is reset when the inner node is deleted). Thus, we have to avoid setting any of the container references of an inner node to null.

5 Conclusion

We think that the NMF solution highlights the advantages model transformations based on synchronization blocks can offer in terms of implicit incremental and bidirectional execution opportunities. On the contrary, the solution also unveils limitations when it comes to synchronizing information represented in tree structures: The necessity to implement the synchronization between the flat collection structure and a tree structure using virtual collections took the largest part of the implementation with relatively little help from the formalism. A better support for such kind of transformation between structures would therefore be highly welcomed.

References

[1] A. Garcia-Dominguez and G. Hinkel, "Truth Tables to Binary Decision Diagrams," in *Proceedings of the 12th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2019) federation of conferences*, ser. CEUR Workshop Proceedings, CEUR-WS.org, 2019.

[2] G. Hinkel and E. Burger, "Change Propagation and Bidirectionality in Internal Transformation DSLs," *Software & Systems Modeling*, 2017.

[3] G. Hinkel, "Implicit Incremental Model Analyses and Transformations," PhD thesis, Karlsruhe Institute of Technology, 2017.

[4] G. Hinkel, "NMF: A multi-platform Modeling Framework," in *Theory and Practice of Model Transformations: 11th International Conference, ICMT 2018, Held as Part of STAF 2018, Toulouse, France, June 25-29, 2018. Proceedings*, accepted, to appear, Springer International Publishing, 2018.

[5] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, no. 3, 2007.

[6] G. Hinkel, R. Heinrich, and R. Reussner, "An extensible approach to implicit incremental model analyses," *Software & Systems Modeling*, 2019.

[7] G. Hinkel, T. Goldschmidt, E. Burger, and R. Reussner, "Using Internal Domain-Specific Languages to inherit Tool Support and Modularity for Model Transformations," *Software & Systems Modeling*, pp. 1–27, 2017.

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.

*ObjectTable*s 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*

In: A. Garcia-Dominguez, G. Hinkel and F. Krikava (eds.): Proceedings of the 12th Transformation Tool Contest, Eindhoven, The Netherlands, 19-07-2019, 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

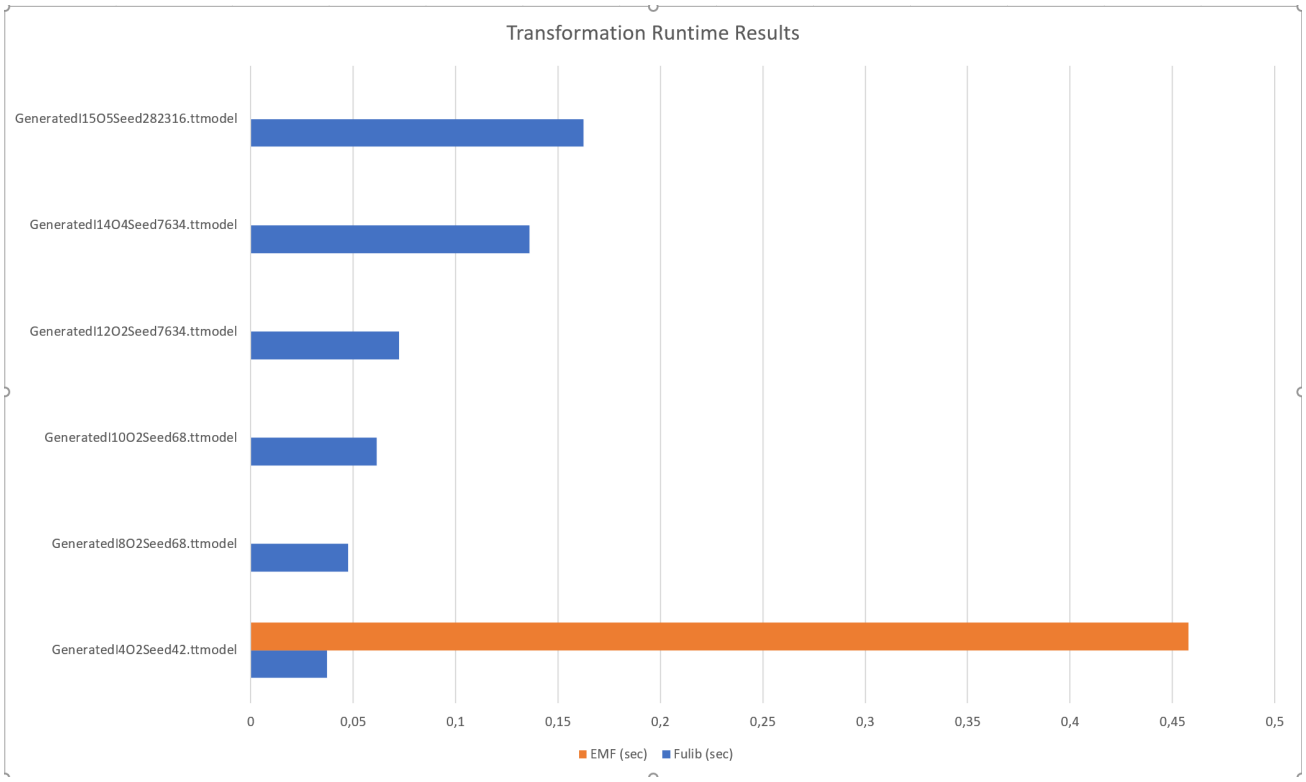


Figure 1: Runtime results

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

YAMTL Solution to the TTC 2019 TT2BDD Case

Artur Boronat
Department of Informatics, University of Leicester, UK
aboronat@le.ac.uk

1 Introduction

In this paper, we present a solution for the TTC'19 TT2BDD case[1], where a set of boolean functions represented in a truth table with several input ports and several output ports is to be represented using binary decision diagrams (BDDs) [Bry86]. Our solution[2] is implemented using YAMTL [Bor18], an internal DSL of Xtend[3] that enables the use of model-to-model transformation within JVM programs, and produces a target model containing one reduced ordered BDD (ROBDD) [Bry86, Bry92] per boolean function[4], whose signature is given by the input ports and by one output port in the source truth table. In the rest of the paper, we discuss relevant aspects of our solution and we evaluate it according to the criteria proposed in the case.

2 Solution

The YAMTL solution is implemented with a model transformation that is executed in two stages, as illustrated in Fig. 1. In the first stage, the truth table is represented as a set of boolean expressions in disjunctive normal form (DNF), one per output port. In the second stage, the target model is built and a ROBDD is build from each of the boolean expressions.

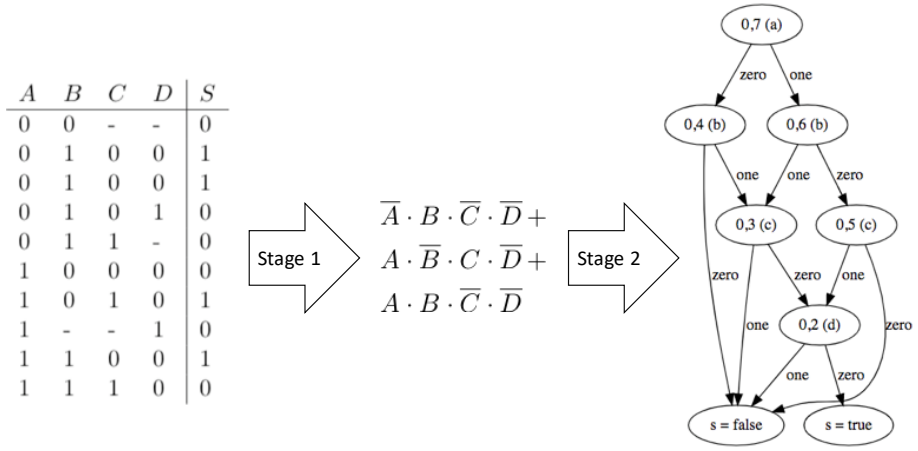


Figure 1: Outline of Solution

In the following subsections, the representation of boolean expressions used in the solution is presented. Then the two stages of the transformation are explained.

In: A. Garcia-Dominguez, G. Hinkel and F. Krikava (eds.): Proceedings of the 12th Transformation Tool Contest, Eidhoven, Netherlands, 19-07-2019, published at http://ceur-ws.org

[1] `https://www.transformation-tool-contest.eu/2019-tt2bdd.pdf`

[2] Available at `https://github.com/arturboronat/ttc2019-tt2bdd/tree/master/solutions/EMFSolutionYAMTL/`

[3] In the rest of the paper, some familiarity with Xtend is assumed. In particular, no effort has been made at representing its dialect for handling collections as pseudocode.

[4] The upper bound of the reference `BDD::root` in the graph BDD metamodel (`BBDv2.ecore`) has been set to `*` to enable this.

2.1 Boolean Expressions and their Reduction

Boolean expressions are represented in disjunctive normal form (DNF), i.e., as a disjunction of conjunctive clauses. Each conjunctive clause is represented using a list of integers, where the index of a member identifies the input port and where the value `0` represents a negated port $\overline{A}$, and the value `1` represents a port. In the source truth table, a row that does not include any assignment for a port represents a set of rows, corresponding to rows containing all possible assignments for such port. This fact has been accounted for by including tautologies of the form $A+\overline{A}$ in conjunctive clauses, which are represented with the value `2`, and the actual conjunctive clauses can be obtained using the + distributive law. For example, the first row in the truth table of Fig. 1 is captured by the formula $\overline{A}\cdot\overline{B}$, which is represented as `[0,0,2,2]`. For each output port, a boolean expression is constructed as a disjunction of conjuntive clauses that are valid, i.e. whose output port value is `1`. The disjunction is represented by using a list of conjunctive clauses. The example in Fig. 1 shows the boolean expression obtained for the given truth table, where the expression $\overline{A}\cdot\overline{B}$ is not included. `dnfMap` represents a global map of boolean expressions, indexed by output ports.

An important step in the construction of BDD graphs used in our solution consists in computing a successor after replacing a port with either `0` or `1`. The reduction step, implemented in the listing below, involves replacing each port in a clause for its value with a subsequent cancellation of clauses that become invalid (line 3), when the port is negated $\overline{A}$ and the value is `true` or when the port A is not negated and the value is `false`. Cancellation is performed by removing the corresponding clause from the list (line 3). When a tautology of the form $A+\overline{A}$ is present in a clause, representing several clauses, then a conjunctive formula is cancelled (line 4). For example, if the value is `false`, then the formula with $\overline{A}$ would be preserved. When an expression `dnf` is reduced for all of its input ports, then each remaining element in the list `dnf` represents a list of assignments for input ports that satisfies a clause, thereby denoting that the expression `dnf` is valid. If no elements are left, then the boolean expression is not valid for the list of assignments used in reduction steps.

```
1 def reduce(List<List<Integer>> dnf, Port x, Integer value) {
2   val i = x.owner.ports.indexOf(x)
3   dnf.removeIf([r | r.get(i) != value])
4   dnf.filter[r | r.get(i)==2].forEach[r | r.set(i, value.complement)]
5   dnf
6 }
```

2.2 Stage 1: Obtaining Boolean Expressions in DNF

In the first stage, the rows of the source truth table are processed with a high priority rule `Row`, which builds a conjunctive clause from the values of the input ports in the row and, for each output port, completes the corresponding DNF expression in `dnfMap` if it is valid.

In YAMTL, a rule consists of an `in` pattern matching elements of the source model, which may contain filters, and of an `out` pattern, creating elements in the target model. Rule `Row` applies to rows in the truth table and creates an assignment in the target model. This rule does not have any side effect in the target model because it is declared as `transient` but it does initialize the `dnfMap` containing a DNF boolean expression per output port, as explained in the previous section. The YAMTL operator `fetch` at the start of the out element action (line 3) retrieves the value of the matched object `r`.

```
1 new Rule('Row').priority(1).transient // input cells are assumed to appear before output cells
2   .in('r', TT.row).build()
3   .out('a', BDD.assignment, [ val r = 'r'.fetch as Row
4     val portList = new ArrayList<ttc2019.metamodels.tt.Port>(r.owner.ports)
5     val oPortList = portList.filter[it instanceof ttc2019.metamodels.tt.OutputPort]
6       .map[it as ttc2019.metamodels.tt.OutputPort]
7     if (dnfMap.empty) // initialize list of boolean expressions, one per output port
8       for (oPort: oPortList) dnfMap.put(oPort, newArrayList)
9     val List<Integer> clause = newArrayList
10    for (c: r.cells) // each row yields a dnf formula
11      if (c.port != portList.head) { // a port is not set in the row
12        clause.add(2) // considered as a tautology: c.port or not(c.port)
13      } else {
14        portList.remove(c.port)
15        if (c.port instanceof ttc2019.metamodels.tt.InputPort)
16          clause.add(c.value.toInt) // false.toInt=0, true.toInt=1
17        else // when the dnf is valid, it is added to the corresponding expression
18          if (c.value) dnfMap.get(c.port).add(clause)
19      }
20  ]).build()/* for out element */.build()/* for rule */
```

2.3 Stage 2: Building ROBDDs

In the second stage, the BDD model with ports is created with the rules `TableToBDD`, `InputPort` and `OutputPort`, which are not transient. The class `AuxStruct` declares three auxiliary data structures: `nodeIndex`, defined for indexing nodes in ROBDDs; `nodeMap` for obtaining the index of a given ROBDD node; `treeMap` for caching nodes by the name of the input variable and by the indexes of the zero and one succesor nodes, `zeroIndex` and `oneIndex` resp., in order to avoid duplication.

The rule `TableToBDD` builds the root object of the model and creates input and output ports in the BDD model via the rules `InputPort` and `OutputPort` respectively. The YAMTL operator `fetch` at the start of the out element action (line 4) retrieves the value of the matched truth table `tt` and of the created out BDD object `bdd` (and similarly for the other rules). The YAMTL operator fetch, in lines 5-6, resolves a list of references to elements in the source model by returning a list of references to elements in the target model, as they are transformed by other rules. In particular, YAMTL will automatically find that these references are resolved with the rule `InputPort` in line 5 and by the rule `OutputPort` in line 6. The rule `TableToBDD` also uses the helper `build`, which returns the root tree nodes of each ROBDD corresponding to each output port in the truth table.

```
1 new Rule('TableToBDD')
2   .in('tt', TT.truthTable).build()
3   .out('bdd', BDD.BDD, [
4     val tt = 'tt'.fetch as TruthTable; val bdd = 'bdd'.fetch as BDD
5     bdd.ports += tt.ports.fetch('bdd_p','InputPort') as List<InputPort>
6     bdd.ports += tt.ports.fetch('bdd_p','OutputPort') as List<OutputPort>
7     bdd.root += dnfMap.build(bdd)
8   ]).build()
9 .build(),
10
11 new Rule('InputPort')
12   .in('tt_p', TT.inputPort).build()
13   .out('bdd_p', BDD.inputPort, [
14     val tt_p = 'tt_p'.fetch as tt.InputPort; val bdd_p = 'bdd_p'.fetch as InputPort
15     bdd_p.name = tt_p.name
16   ]).build()
17 .build(),
18
19 new Rule('OutputPort')
20   .in('tt_p', TT.outputPort).build()
21   .out('bdd_p', BDD.outputPort, [
22     val tt_p = 'tt_p'.fetch as tt.OutputPort; val bdd_p = 'bdd_p'.fetch as OutputPort
23     bdd_p.name = tt_p.name
24   ]).build()
25 .build()
```

For each boolean expression in `dnfMap`, the helper `build` initializes the auxiliary data structures in an `AuxStruct` instance and calls the helper `build` for that particular expression (`entry.value`) returning the root of the generated ROBDD, which is then added in the `rootTreeList` (line 6 below).

```
1 def build(Map<tt.OutputPort,List<List<Integer>>> dnfMap, BDD bdd) {
2   var List<Tree> rootTreeList = newArrayList
3   val iPortList = bdd.ports.filter[it instanceof InputPort].map[it as InputPort].toList
4   for (entry: dnfMap.entrySet) {
5     val oPort = bdd.ports.findFirst[it.name==entry.key.name] as OutputPort
6     rootTreeList.add(entry.value.build(iPortList, oPort, new AuxStruct))
7   }
8   rootTreeList
9 }
```

The helper `build` for a particular boolean expression `dnf`, in the listing below, takes the next input port from `iPortList` to be processed (line 2), and builds the ROBDD for the assignments `0` (in line 5) and `1` (in line 6) using the helper `buildTree`, which returns the root of the corresponding ROBDD and indexes the nodes of the ROBDD. When the indexes of the root nodes for both branches coincide, then the two branches are the same and no new node needs to be added (line 11-12). Otherwise, a new node is created and cached if it has not been indexed by input port name and indexes of successors (lines 16-19). When the node to be created has been indexed already (the `triple` in line 15 is not null), then the subtree has already been created and it does not need to be duplicated. The helper `createSubtree` instantiates the class `Subtree` and initializes its variable and children, indexing it in `treeMap`.

```
def build(List<List<Integer>> dnf, List<InputPort> iPortList, OutputPort oPort, AuxStruct aux) {
  val iPort = iPortList.head
  val portTail = iPortList.tail.toList
  // create successors
  var tree0 = dnf.clone().buildTree(iPort, portTail, oPort, 0, aux)
  var tree1 = dnf.clone().buildTree(iPort, portTail, oPort, 1, aux)
  // create tree by adding root
  var Tree tree
  val zeroIndex = aux.nodeIndex.get(tree0)
  val oneIndex = aux.nodeIndex.get(tree1)
  if (zeroIndex == oneIndex) { // avoid redundant tests
    tree = tree0
  } else { // create tree if not yet indexed by port name, zero index and one index
    val triple = new ImmutableTriple(iPort.name, zeroIndex, oneIndex)
    tree = aux.treeMap.get(triple)
    if (tree === null) {
      tree = iPort.createSubtree(tree0, tree1, aux) // creates a subtree with root x and two children
      aux.treeMap.put(triple,tree)
    }
  }
  tree
}
```

The helper `buildTree`, in the listing below, reduces the boolean expression `dnf` by substituting `value` for the input port `iPort` (line 3), obtaining `reducedDNF`, and continues processing the next input port by calling `build` for `reducedDNF` (line 5). If the list of remaining input ports `portTail` is empty, the helper returns a `zero` leaf (line 8) when there are no assignments for the expression `dnf`, i.e. the list is empty meaning that the DNF is not satisfiable, and a `one` leaf (line 10) when there are assignments, i.e. the list `dnf` contains the assignments for the input ports for which the expression is valid. The `createLeaf` creates a `Leaf` instance, initializing its value and indexing the node.

```
def Tree buildTree(List<List<Integer>> dnf, Port iPort, List<InputPort> portTail, OutputPort oPort, Integer
    value, AuxStruct aux) {
  var Tree tree
  val reducedDNF = dnf.reduce(iPort, value) // applies substitution value/x and cancels invalid dnfs
  if (!portTail.empty) { // there are still more variables to be reduced
    tree = reducedDNF.build(portTail, oPort, aux)
  } else { // no variables left
    if (reducedDNF.empty) // no DNFs left in the expression: the result is 0
      tree = oPort.createLeaf(0, aux) // helper that creates leaf node for oPort and indexes it at 0
    else // the result is 1
      tree = oPort.createLeaf(1, aux) // helper that creates leaf node for oPort and indexes it at 1
  }
  tree
}
```

3 Evaluation and Conclusions

Below we discuss the criteria proposed to evaluate the solution, which we conclude by highlighting the aspects of YAMTL that have been exploited in this solution.

Input Model		ATL (tree)				YAMTL			
input	output	size	init	load	run	size	init	load	run
4	1	32	256.9	118.8	457.6	16	247.0	110.5	59.8
4	2	70	253.0	124.5	594.5	30	245.4	126.1	66.9
8	2	1034	249.9	220.0	53.7′	168	245.1	252.8	136.2
10	2	4108	252.1	375.7	1330.8′	503	256.7	404.9	222.0
12	2					1489	242.5	914.1	397.5
14	4					9193	248.9	2.3′	1.1′
15	5					21592	244.5	4.3′	2.5′

Table 1: Model element cardinalities and execution times in ms. (unless stated ′ for s.)

3.1 Correctness

As explained above, each output port in the final BDD model corresponds to the output of a boolean function, which is represented as a separate ROBDD. The correctness of our solution has been checked with a generalization of the proposed graph validator, which runs the validator for each of the ROBDDs that have been generated and that are available under `BDD::root` (with upper bound `*`).

3.2 Completeness

The solution can be executed for all of the sample input truth table models provided, including up to 15 input ports and up to 5 output ports.

3.3 Performance

For input models, Table 1 shows the number of *input* ports and *output* ports. The table also shows the size of the resulting BDD model (object cardinality) and execution times for each of the phases (*init*ialization, *load*, *run*) both for the reference solution (tree metamodel) and for the YAMTL solution.

For the asymptotic analysis of our solution, we consider n input ports and m output ports in the source truth table. The algorithm in our solution is split in two parts. First, the truth table is linearly processed in order to build boolean expressions, which involves up to n^2 rows consisting of n input cells, that is $\mathcal{O}(n^3)$. Second, each boolean expression consisting of n input ports is simplified, once for each output port, that is $n \times m$. Simplification involves reducing each port in all clauses of a DNF expression. In case of a tautology, this involves making substitutions in n^2 clauses! Therefore, the worst time complexity of the second part is $\mathcal{O}(m \times n^3)$, and the worst time for the overall solution is $\mathcal{O}(n^3) + \mathcal{O}(m \times n^3)$. However, as shown in Fig. 1, tautologies are a rare case and we can assume that the second part will tend towards the best time complexity, which is $\mathcal{O}(m \times n)$, during the reduction of each boolean expression. Hence, average performance is increasingly dominated by the processing of the truth table in the first part as the number of input ports grow and, in practice, our solution tends to be linear in terms of the number of cells, corresponding to input ports, in the source truth table, that is $\mathcal{O}(n^3)$.

3.4 Optimality

Our BDD models correspond to lists ROBBDs, one per output port, where all paths through the graph respect the linear order of input ports, as defined in the input table, and all nodes are unique (no two distinct nodes have the same variable name and zero and one successors) and contain no redundant tests (no variable node has identical zero and one successors). The result is a compact representation of a boolean function, as can be observed in the size column of the solutions in Table 1.

3.5 Conclusion and Research Direction

The solution presented is correct and complete w.r.t. the evaluation criteria provided in the case. When comparing it with the reference ATL transformation (on the tree metamodel), in Table 1, the transformation shows good performance and produces smaller output models. Moreover, the solution presented showcases several features of YAMTL: interoperability with Java data structures via Xtend; application of rules in stages using priorities; declarative rules with side effects, one of which is transient.

References

[Bor18] Artur Boronat. Expressive and efficient model transformation with an internal dsl of xtend. In *Proceedings of the 21th ACM/IEEE International Conference on MoDELS*, pages 78–88. ACM, 2018.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

Transforming Truth Tables to Binary Decision Diagrams using Relational Reference Attribute Grammars

Johannes Mey, René Schöne, Christopher Werner and Uwe Aßmann
{first.last}@tu-dresden.de

Software Technology Group
Technische Universität Dresden

Abstract

The TTC 2019 case describes the computation of a decision diagram from a truth table. In this paper, we present a solution which uses relational reference attribute grammars to represent both input and output model. To transform the former into the latter, we implemented different strategies and present in detail an algorithm using a reduced order binary decision diagram utilizing higher-order attributes. We further present transformation times and the number of decision nodes in the result model showing the feasibility of our approach.

1 Introduction

In the TTC 2019 case, a given truth table has to be transformed into a logically equivalent binary decision diagram. Both truth table and binary decision diagram (BDD) are two different ways to represent a function over Boolean tuples, where on the one hand truth tables can be seen as a set of assignments, each given as a row of the table. On the other hand, a binary decision diagram represents this function as a directed acyclic graph (DAG) with a single source. The nodes of the DAG are partitioned into decision nodes and leaf nodes. Decision nodes refer to input variables and have edges annotated with either 0 or 1, representing the corresponding decision. Leaf nodes are annotated with an output tuple.[1]

2 Background

Attribute Grammars [Knu68] use a context-free grammar for the declarative definition of an abstract syntax tree (AST) and attributes to define analyses on this tree. In [Hed00], this concept was extended to Reference Attribute Grammars (RAGs), which allow attributes to compute nodes of the AST, effectively enabling the definition of graphs instead of trees. Our solution uses Relational Reference Attribute Grammars [MSH+18], which in turn extend RAGs with first-class relations, i.e, edges between nodes in the AST, to provide an easy definition of those relations and to enable bidirectional relations. Furthermore, we use an advanced feature of RAGs, nonterminal attributes [VSK89]. These are attributes computing a new subtree, which after computation is treated like a nonterminal. Using them, implicit knowledge of the tree can be manifested and later be reused. In our implementation, we use *JastAdd* [EH07], a RAG system to define both grammar and attributes. The grammar is specified using a BNF syntax with inheritance and relations. Every nonterminal defined there is compiled to a Java class with accessors for its children, attributes, and relations. Attribute definitions are specified using a Java-based DSL and are woven into the Java class of the nonterminal they are defined in. As described in [MSH+18], a preprocessor transforms the relations into basic grammar rules and special accessors.

[1] BDDs are usually specified for one output variable. Otherwise, *shared BDDs* with multiple source nodes can be used [MT12].

In: A. Garcia-Dominguez, G. Hinkel and F. Krikava (eds.): Proceedings of the 12th Transformation Tool Contest, Eindhoven, The Netherlands, 19-07-2019, published at http://ceur-ws.org

```
LocatedElement ::= <Location:String>;
TruthTable : LocatedElement ::= <Name:String> Port:Port* Row:Row* ;  // The start symbol of this grammar

abstract Port : LocatedElement ::= <Name:String>;                   // Port is an abstract node that cannot be instantiated
InputPort : Port;                                                   // InputPort and OutputPort are a special ports, i.e., inheriting
OutputPort : Port;                                                  //  name and location from it
Row : LocatedElement ::= Cell:Cell*;
Cell : LocatedElement ::= <Value:Boolean>;                          // The cell has a intrinsic attribute, in this case
                                                                    //  the boolean value of the cell
rel Port.Cell* <-> Cell.Port;                                       // A bidirectional relation stating, that a cell refers to a port
```

Listing 1: Grammar for a truth table in *JastAdd* syntax

```
BDT ::= <Name:String> Port:BDT_Port* Tree:BDT_Tree;
abstract BDT_Tree;
BDT_Leaf:BDT_Tree ::= Assignment:BDT_Assignment*;
BDT_Subtree:BDT_Tree ::= TreeForZero:BDT_Tree TreeForOne:BDT_Tree;

abstract BDT_Port ::= <Name:String>;
BDT_InputPort : BDT_Port;
BDT_OutputPort : BDT_Port;
BDT_Assignment ::= <Value:boolean>;

rel BDT_InputPort.Subtree* <-> BDT_Subtree.Port;
rel BDT_OutputPort.Assignment* <-> BDT_Assignment.Port;

// relations to TruthTable model
rel BDT.TruthTable -> TruthTable;
rel BDT_InputPort.TruthTableInputPort -> InputPort;
rel BDT_OutputPort.TruthTableOutputPort -> OutputPort;
rel BDT_Leaf.Row* -> Row;
```

Listing 2: *JastAdd* grammar for Binary Decision Tree

```
BDD ::= <Name:String> Port:BDD_Port* Tree:BDD_Tree*;
abstract BDD_Tree;
BDD_Leaf:BDD_Tree ::= Assignment:BDD_Assignment*;
BDD_Subtree:BDD_Tree;
abstract BDD_Port ::= <Name:String>;
BDD_InputPort : BDD_Port;
BDD_OutputPort : BDD_Port;
BDD_Assignment ::= <Value:boolean>;
rel BDD.Root -> BDD_Tree;
rel BDD_Subtree.TreeForZero <-> BDD_Tree.OwnerSubtreeForZero*;
rel BDD_Subtree.TreeForOne <-> BDD_Tree.OwnerSubtreeForOne*;
rel BDD_InputPort.Subtree* <-> BDD_Subtree.Port;
rel BDD_OutputPort.Assignment* <-> BDD_Assignment.Port;
// relations to TruthTable model
rel BDD.TruthTable -> TruthTable;
rel BDD_InputPort.TruthTableInputPort -> InputPort;
rel BDD_OutputPort.TruthTableOutputPort -> OutputPort;
rel BDD_Leaf.Row* -> Row;
```

Listing 3: Grammar for a Binary Decision Diagram

```
PortOrder;
rel TruthTable.PortOrder -> PortOrder ;
rel PortOrder.Port* -> InputPort;
```

Listing 4: Grammar for *PortOrder*

```
// TruthTable has two NTA defined:
syn PortOrder TruthTable.getNaturalPortOrder() = //...
syn PortOrder TruthTable.getHeuristicPortOrder() = //...
```

Listing 5: NTA definition for *PortOrder*

3 Computing a Binary Decision Diagrams with Relational RAGs

In this section, we show how truth tables, BDTs, and BDDs can be modelled using relational RAGs and, in particular, how relations support the transformations and traceability between the models. Initially, the case description demanded a binary decision *tree* (BDT) and not a binary decision diagram (BDD). Therefore, we present different configurable algorithms to transform a truth table into either a BDT or a BDD. While the outputs of all provided transformations are semantically equivalent for one given input, they have different characteristics regarding both the runtime of the transformation and the properties of the resulting diagrams.

3.1 Describing Conceptual Models

Many conceptual models can be described by relational RAGs. This is possible, because many model specification languages, e.g., Ecore, require models to have a *containment hierarchy*, and thus a spanning tree. Listing 1 shows the grammar of the truth table that corresponds to the provided model. Since the grammar specification rules of *JastAdd* use concepts like inheritance and abstract types, all given Ecore models can easily be transformed into relational RAGs. The resulting BDT and BDD grammars are shown in Listings 2 and 3. While some concepts like *Port* potentially could be reused between the models, we refrained from doing so, because those concepts do not have exactly the same definition in all models.[2]

Important aspects when dealing with several trees at the same time defined by relational RAGs are the modularity of their specification and the reachability between those trees. RAGs as specified by *JastAdd* are inherently modular: Both grammar and attributes can be split into aspect files. Therefore, both truth table, decision tree and -diagram models are described by one relational grammar, but specified in different modules. Since the truth table grammar shown in Listing 1 has no relations to the other models, it can be used independently. On the other hand, the diagram models have a relation to the truth table, as shown in Listing 2. Note that this is not required, but a design decision, since those relations enable traceability between the models.

[2]For example, a *Port* within a truth table is a *LocatedElement*, while one in a BDD is not. In other cases, the types may be the same, but the relations between them are not.

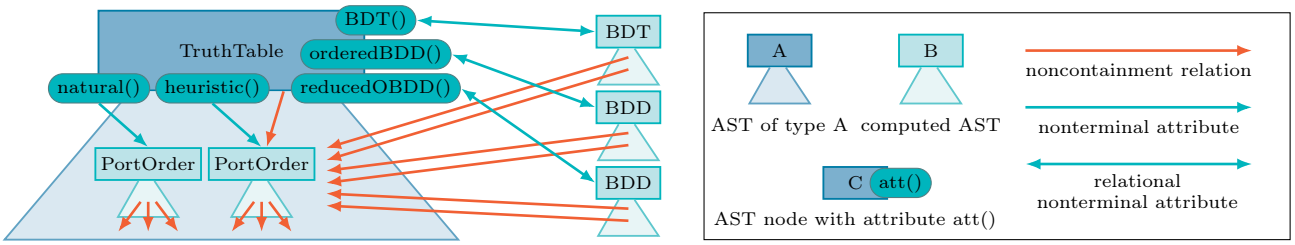


Figure 1: Elements of the transformation and their relations

3.2 Computing Models

A model transformation can be seen as the computation of a target model using information of a source model. RAGs provide a mechanism for that: attributes that compute a tree of the grammar they are defined in, called *higher-order* or *nonterminal attributes (NTAs)*. There are two types of NTAs, those that describe a *subtree* of an AST and those that describe a new tree. In both cases, non-containment relations added by *relational* RAGs provide the means to link these (sub-)trees to the original tree. Figure 1 shows the structure of the trees used for the case. The only given tree is the *TruthTable* tree, containing two subtrees of the type *PortOrder* shown in Listing 4, defined by NTAs from Listing 5. The results of the transformation are stored in separate trees of type *BDT* and *BDD*, computed by the relational NTAs *BDT()*, *orderedBDD()* and *reducedOBDD()*. Note that different NTAs may return different instances of the same *type*. In addition to the relations computed by NTAs, there may be other *intrinsic* relations, such as the relation that selects one of the two provided *PortOrder*s. These relations can also link nodes in computed subtrees to nodes in the originating tree.[3]

In the following, we focus on one transformation into reduced ordered BDDs using the NTA *reducedOBDD()*.

3.3 Computing an Ordered Binary Decision Diagram

Constructing an optimal BDD is a computationally hard, but, since there are many real world applications that use large BDDs, appropriate simplifications and efficient heuristics exist. One commonly used simplification of BDDs are ordered BDDs (OBDD), in which the order of input variables[4] in all paths is identical, allowing a layering of the graph. Given an OBDD, two simple reduction rules are applied to reduce the number of nodes; the result of such a reduction process is a reduced OBDD. In an OBDD, the minimal diagram for a given order can be computed efficiently, however, the computation of the optimal order is still NP-hard [MT12, p. 139ff.].

We follow the definition of an OBDD given in [MT12] extending the problem to several output variables as mentioned in Section 1 as follows:

- An OBDD with m input and n output variables has at most 2^n leaves, each with a different assignment function $f : V_o \to \{0, 1\}$ with V_o as the set of output variables.
- There is an order $<_\pi$ for input variables, such that on an edge from one node referring to input variable x to a node referring to variable y it holds that $x <_\pi y$.

Conceptually, we split the construction of a reduced OBDD in three stages: the computation of a port order, the construction of a perfect tree, and its reduction. Listing 6 shows the synthesized attribute that performs this process. First, a BDD node is constructed and the relations to the truth table and its variables are established (lines 2–5). Then, the leaves are constructed by a helper function and added to the diagram (line 6). Afterwards, a port order is retrieved by accessing the noncontainment relation to a *PortOrder* pointing to one of the NTAs that computes a port order. These are defined in a separate grammar aspect shown in Listing 4. Besides the default order defined in line 2 of Listing 5, we provide a heuristic ordering defined in line 3. Since most heuristics in the literature rely on a logical formula as an input, we have chosen to use a simple metric based on the correlation of an input variable to the output vector in the given truth table. Using this port order, the tree is constructed iteratively by adding the path for each input row (Listing 6, lines 9–13). Finally, the reduction is performed in line 14. The employed algorithm and its properties can be studied in [MT12, p. 96ff.]. For a given truth table, there is exactly one minimal OBDD, computed by the algorithm in $\mathcal{O}(d \log d)$ for d decision nodes.

Besides computing and reducing an OBDD, we have also implemented other approaches as shown in Section 5. How to apply those approaches to the given case will be described in the next section.

[3]These relations must not be bidirectional, since the direction from the source model to the computed NTA model would violate the rule that attribute computations may not alter the tree — other than adding the result of the computation.

[4]In literature such as [MT12, RK08], ports are called variables.

```
1  syn BDD TruthTable.reducedOBDD() {
2    BDD bdd = this.asBDD();                                  // Construct an empty BDD with trace link to truth table
3    bdd.setName(getName());                                  //   and copy the name from the truth table
4    for (Port port: this.getPortList())                      // For each port
5      bdd.addPort(port.asBDDPort());                         //   the according BDD variant of the port is created and linked back
6    for (BDD_Leaf leaf: constructLeaves()) bdd.addTree(leaf); // Add leaves
7    PortOrder portOrder = getPortOrder();                    // Obtain the port order by accessing a non-containment relation
8                                                             //   to the selected nonterminal attribute
9    BDD_Subtree root = new BDD_Subtree();                    // Create root node, set its port to the first in the port order,
10   root.setPort(bdd.bddInputPort(portOrder.getPortList().get(0))); //   add it to the BDD, and specify it as the root
11   bdd.addTree(root);
12   bdd.setRoot(root);
13   for (Row row : getRowList()) insertRow(bdd, root, row, 0);  // Fill the BDD by adding all defined paths using a helper function
14   bdd.reduce();                                            // Perform the reduction by calling a helper function
15   return bdd;
16 }
```

Listing 6: Relational non-terminal attribute to create an ordered BDD

```
1 syn int BDT_Tree.decisionNodeCount();
2 eq BDT_Subtree.decisionNodeCount() = 1 + getTreeForZero().decisionNodeCount() + getTreeForOne().decisionNodeCount();
3 eq BDT_Leaf.decisionNodeCount() = 0;
```

Listing 7: Computation of the number of decision nodes in the BDT

4 A Dynamic Transformation Toolchain

To perform the transformation, we follow the process outlined in Figure 2. We will show, how our approach supports variability that enables reuse and the combination of different transformation approaches and how relational RAGs support traceability between models and help in analysing these models.

As *JastAdd* is not based on Ecore, the meta model of EMF [SBMP08], we can not directly use the given input models. Instead, we translated the given metamodel into the grammar shown in Listing 1 and built a hand-written XMI parser constructing an AST according to this grammar. However, relational RAGs provide some mechanisms to reduce the implementation overhead and simplify this process.

While parsing an XMI file is rather straightforward, the resolution of the XMI references is not. In attribute grammars, name analysis is well-supported and a frequently used application. Relational RAGs provide an additional method to defer the name resolution after the parsing while still allowing the result of the attribute-computed name resolution to be stored as a non-containment relation. Once the truth table is parsed, the transformation is performed. The first step is to create an additional subtree with a *PortOrder*, which can later be used to create ordered BDTs and BDDs. This is also the first configurable step of the process, since the algorithm for the computation of the *PortOrder* can be switched. Afterwards, BDTs or BDTs are created by different attributes. In the case of the OBDD, the reduction can be seen as an additional step of the computation. Since all variants are independent trees, it is possible to create several variants at the same time. Each created variant contains trace links into the truth table model that are created during construction (cf. Listing 2, lines 15 to 18). Then, the results are validated and different metrics described in Section 5 are computed. One example for a metric is the number of decision nodes in Listing 7. Finally, the resulting BDT or BDD is serialized to XMI. Again, this step requires attributes to compute the references within the file, i.e., XMI path expressions. This whole process is embedded into the provided benchmarking framework[5] and evaluated in the next section.

[5]The implementation can be found at `https://git-st.inf.tu-dresden.de/ttc/bdd`

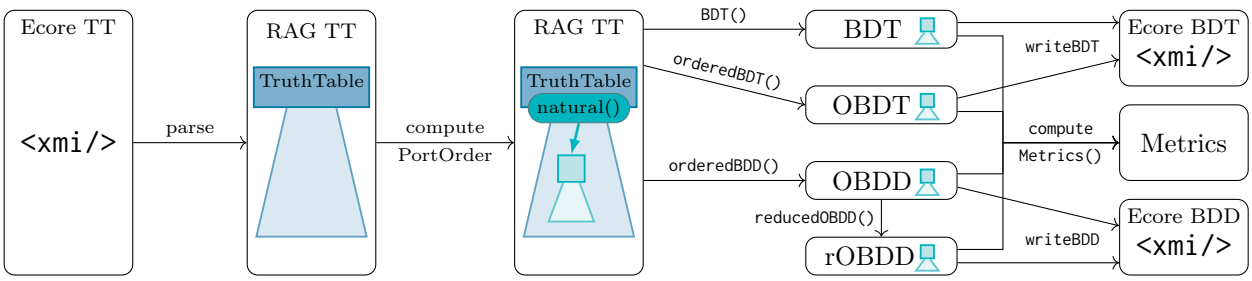
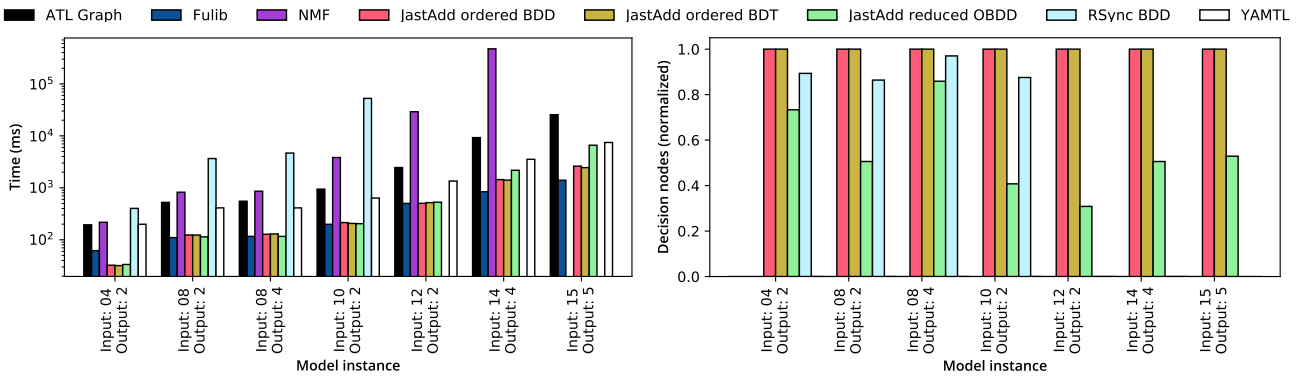


Figure 2: Transformation process and temporary artefacts



(a) Total time

(b) Number of decision nodes (normalized)

Figure 3: Evaluation results

5 Evaluation

The evaluation consists of two parts. First, the transformation and its properties are discussed with a focus on conciseness, modularity, and reuse, then its performance and the quality of the target models are described.

5.1 Properties of the Transformation

Using relational RAGs, the presented models can be specified concisely. As shown in Listings 1 to 4, the grammars for truth table, BDT and BDD comprise few lines of code. The specification of the transformation using the attributes of *JastAdd* proves to be a good combination of imperative code (which is beneficial for such complex transformation algorithms as, e.g., for the OBDD reduction) and efficient attribute-supported tree navigation. Additionally, attributes can help by checking both the *correctness* and different *metrics* of the obtained model.

An important aspect of the presented solution is modularity and the opportunities for reuse that stem from it. Even though, technically, all models are combined in one large grammar, this grammar can be used in independent modules consisting of grammar fragments and accompanying attributes. The example of the *PortOrder* extension shows how additional analysis modules can simply be added to the grammar. Using relations, entire new models can be integrated and used by existing models. On the attribute level, the definition of attributes in *aspects* also helps to combine and reuse separate parts of the transformation system. Furthermore, relational RAGs allow traceability. Relations between models can be established that, e.g., show which rows have taken part in the creation of a *Leaf* in a BDD.

5.2 Performance and Quality

To evaluate the runtime performance, the measurements of the TTC organizers were considered. All measurements were performed on a Google Cloud Compute Engine `c1-standard-4` machine with 16GiB of RAM and an 30GB SSD, using a Docker image.[6] Each configuration was run ten times. In addition to time measurements, we added a number of result quality metrics computed by attributes, namely the number of nodes of the two different types and the minimum, maximum, and average path length.

Figure 3 shows our evaluation results for the seven provided input models depicting total runtime and the number of decision nodes, i.e., instances of type *Subtree*, normalized to the greatest number among all variants.[7] For the measurement, we included the following variants of our approach: An ordered BDT variant generating the tree iteratively (*ordered BDT*), an ordered BDD generated iteratively (*ordered BDD*) and a reduced variant of the latter, using the algorithm described in Section 3.3 (*reduced OBDD*). For the last variant, we used the order determined by the heuristic. These solutions are compared to the other contributions to the TTC that integrate into the benchmark infrastructure. As shown in Figure 3a, all variants display a good performance compared to the other solutions; the only solution that is considerably faster for larger problems is *Fulib*, which benefits from being a pure Java solution optimized for the input data of the benchmark. However, relational RAGs do not require any initialization other than loading the required Java classes, resulting in only a small overhead compared to *Fulib* that we assume is due to the larger memory footprint of *JastAdd*. Comparing the execution times of our approaches, the non-reduced implementations show a very similar performance, while the

[6] `https://github.com/TransformationToolContest/ttc2019-tt2bdd/blob/master/Dockerfile`

[7] Since a variant is included that creates a perfect tree, this number is $2^m - 1$ where m is the number of input variables.

reduced OBDD takes up more time for the largest model, due to the final reduction step.While the benchmark only measured resident memory before and after the transformation, the figures for this stayed below 1GB for all variants and scenarios and are thus lower than for all other solutions except *Fulib*. Looking at the number of decisions in the final result, there are three classes of approaches. All non-reduced OBDD variants generate the largest number of decision nodes, as they always produce the full tree. The algorithm used in the case description reduces the nodes in the tree by 2% to 14% compared to the full tree. When applying the reduction algorithm to an ordered BDD, 14% to 69% of the decision nodes in the full tree are removed.

6 Conclusion and Future Work

We have shown how to apply Relational Reference Attribute Grammars to the problem of transforming truth tables into (ordered) binary decision diagrams. This included defining a suitable grammar for both truth tables and BDDs, parsing the given input models, computing a BDD in different ways while reusing intermediate results, and finally printing the result to the required XMI format. We used the *JastAdd* system to implement our solution and were able to create several configurable transformations with good performance compared to most other solutions. As the focus of this contribution was not to create new transformation algorithms, we show how the implementation of those algorithms, metrics, and tracing relations were improved by relational RAGs.

The presented approach demonstrates a manual bidirectional transformation from Ecore-based models to *JastAdd* ASTs and back. While the grammar as well as the parser and printer were hand-written, there is no obvious reason why this could not be automated. Having an automated transformation from Ecore meta-models to grammars and their instances to ASTs would instantly make a huge set of existing models available for efficient RAG-based analysis and is therefore an important direction of research.

Acknowledgements
This work has been funded by the German Research Foundation within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907), the research project “Rule-Based Invasive Software Composition with Strategic Port-Graph Rewriting” (RISCOS) and by the German Federal Ministry of Education and Research within the project “OpenLicht’.

References

[EH07] Torbjörn Ekman and Görel Hedin. The JastAdd system—modular extensible compiler construction. *Science of Computer Programming*, 69(1):14–26, 2007.

[GDH19] Antonio Garcia-Dominguez and Georg Hinkel. Truth Tables to Binary Decision Diagrams. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 12th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2019) federation of conferences*, CEUR Workshop Proceedings. CEUR-WS.org, July 2019.

[Hed00] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2), 1968.

[MSH+18] Johannes Mey, René Schöne, Görel Hedin, Emma Söderberg, Thomas Kühn, Niklas Fors, Jesper Öqvist, and Uwe Aßmann. Continuous Model Validation Using Reference Attribute Grammars. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2018, pages 70–82. ACM, 2018.

[MT12] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design: OBDD-foundations and applications.* Springer Science & Business Media, 2012.

[RK08] Michael Rice and Sanjay Kulhari. A survey of static variable ordering heuristics for efficient bdd/mdd construction. Technical report, 2008.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework.* Pearson Education, 2008.

[VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI ’89, pages 131–145, New York, NY, USA, 1989. ACM.

Applying formal reasoning to model transformation: The Meeduse solution

Akram Idani[1] German Vega[1] Michael Leuschel[2]

[1]Univ. Grenoble Alpes, Grenoble INP, CNRS, LIG. F-38000 Grenoble France
{Akram.Idani, Germa.Vega}@imag.fr

[2]Universitätsstraße 1n. D-40225 Düsseldorf
Michael.Leuschel@uni-duesseldorf.de

Abstract

The TTC'2019 case study deals with a realistic model transformation which generates binary decision diagrams from truth tables. Among other challenges, the contest emphasizes on correctness which motivated us to apply Meeduse, a tool that we developed in order to define formal semantics with animation facilities of Domain Specific Languages (DSLs). This study allowed us to try how far we can push the abilities of a formal method to be integrated within model-driven engineering. The results were positive, and show that Meeduse can be adapted to model transformation which brings to this field formal automated reasoning tools like AtelierB for theorem proving and ProB for model-checking. Meeduse, combined with ProB, provides three strategies: random animation, interactive animation and model-checking. The first strategy runs randomly the transformation rules until it consumes all the truth table rows and then automatically produces the binary decision diagram. The second strategy allows a step-by-step debugging of the transformation rules, and the third strategy is useful for analysing the reachability of some defined states which allows to verify whether unwanted situations may happen or not.

1 Introduction

This paper presents a brief overview of the application of Meeduse to the TTC'2019 case study [2] and gives the lessons learned from this study[1]. First we notice that the tool was conceived in order to define proved executional semantics of domain-specific languages (DSLs) by integrating the formal B method [1] within EMF-based frameworks like XText, Sirius, GMF. Meeduse was recently developed (in 2018) and has had successful applications in the safety-critical domain, especially for railway systems modeling. The reader can refer to [3, 4] for more information about the overall approach of Meeduse. The challenge of the TTC'2019 case study for us, is to define and run model transformations as executional semantics using a well-established formal technique, the B method. This work starts from the following observations:

In: A. Garcia-Dominguez, G. Hinkel and F. Krikava (eds.): Proceedings of the 12th Transformation Tool Contest, Eindhoven, The Netherlands, 19-07-2019, published at http://ceur-ws.org

[1]The proposed solution and demonstration videos can be found at: `https://github.com/meeduse/Meeduse_TTC_2019`.

- We are not experts in circuit design and hence the application is limited to the cases provided in the TTC'2019 call for solutions.
- We provide transformation rules written in a formal language which is assisted by automated reasoning tools; and hence we believe that domain experts may be attracted by our solution. Indeed, translating a truth table (TT) into a binary decision diagram (BDD), has several applications in safety critical systems where formal methods became a strong requirement.
- Our objective is not to search for the most compact BDD, but to show how a formal method assisted by automated reasoning techniques can be applied for the particular field of model transformation.

2 Summary of Meeduse

We advocate for collaborations between the formal methods (FM) community and the model-driven engineering (MDE) community in order to take benefits of their complementarities. The Meeduse[2] tool favors this communication since it makes possible the use of MDE and FM tools together in one unified framework and supports a pragmatic approach for mixing model-driven engineering with a proof-based formal approach. In practice, the tool brings together two technological spaces: EMF for model driven engineering and the B Method [1] for theorem proving and model-checking. The tool is built on top of three components:

(1) *Translator*: this component automatically translates an Ecore meta-model into an equivalent B specification which represents the structure of the meta-model as well as basic operations like constructors, destructors, getters and setters. The resulting B specification can be manually refined by additional invariants and concrete operational semantics. The proof of correctness of the full specification can be performed by AtelierB which is a theorem prover that assists the B method.

(2) *Injector*: this component takes a model conforming to the Ecore meta-model (which can be designed using EMF-based modeling tools like Sirius, GMF, XText, etc.) and produces a specialized B machine derived from the one generated by the Translator component. This component essentially transforms abstract sets (that represent classes in the meta-model) into enumerations representing the concrete instances of the model, and hence allows model-checking over finite domains.

(3) *Animator*: in Meeduse, animation of B specifications is done using the ProB tool [5] which is an open-source model-checker supporting the B method. The component Animator asks ProB to animate B operations and gets the reached state by means of B variable valuations. Then, the Animator translates back these valuations to the initial EMF model resulting in automatic synchronization of the model.

As Meeduse was not initially designed to define model transformation rules, but to define DSL execution semantics, we need to rethink the model transformation problem in terms of operational semantics of an abstract machine. The global strategy consists in reusing the *Translator* component to help automate the writing of the formal specification of the transformation, and apply the Meeduse's *Animator* synchronization capabilities to produce the resulting EMF output model from a given input model.

3 Specification

3.1 Step 1: merging meta-models

The input of Meeduse is the meta-model of a DSL and hence in order to apply the tool for model-to-model transformation, we first need to merge both input and output meta-models into a single one as presented in Figure 1 where the left hand side presents the TT meta-model and the right hand-side presents the BDD meta-model. We suggest that the execution semantics of the transformation follows a consumption/production technique: instances of output classes are created while consuming instances of input classes. In order to keep track of the modeling elements that have been processed by the transformation, we introduce an abstract meta-class `Element` which allows to gather modeling elements consumed by the transformation. This class introduces an attribute `name` to identify processed elements. We also introduce a Boolean attribute `selected` inside class Cell in order to mark cells being processed by the transformation. Having defined this merging meta-model, we are ready to start thinking about the formalization of the transformation.

[2]Meeduse: Modeling Efficiently EnD USEr needs.

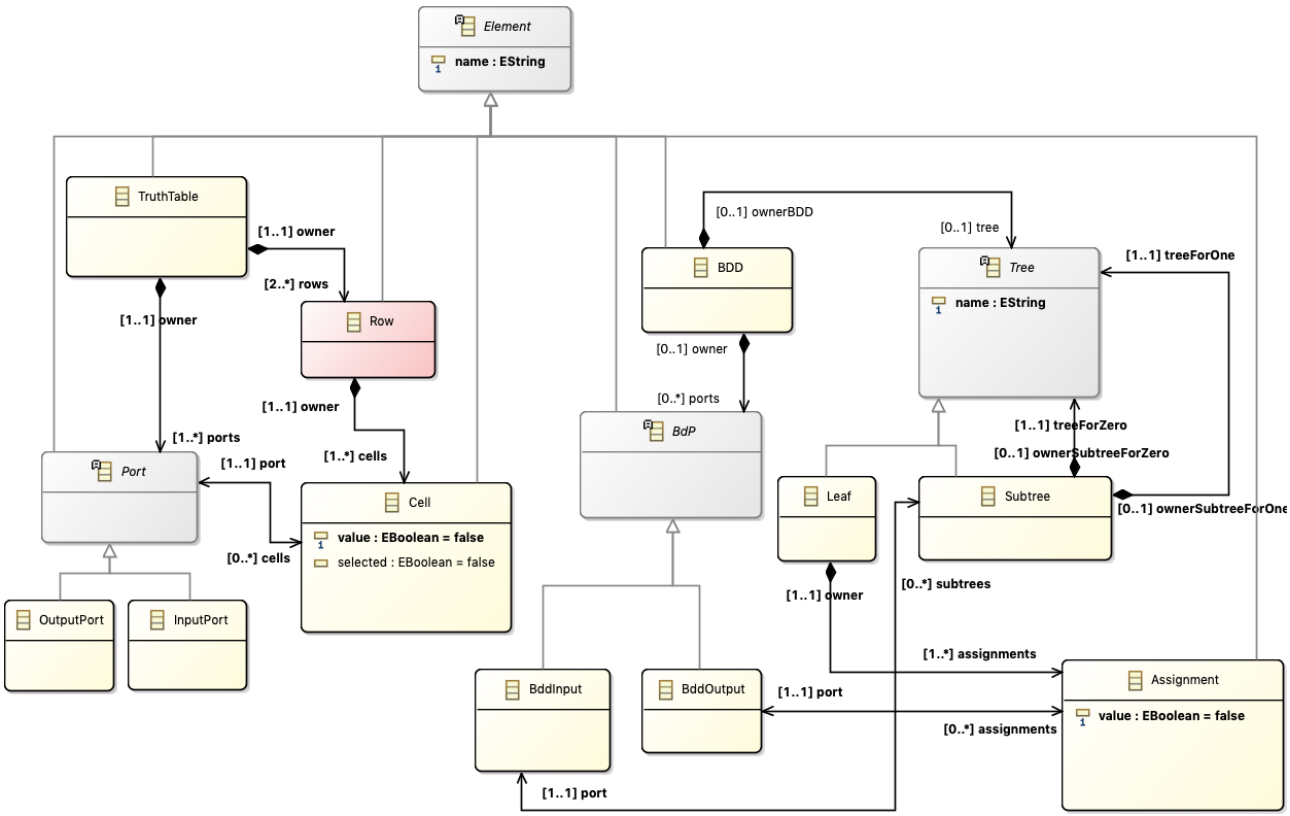


Figure 1: Merging meta-models

3.2 Step 2: generation of the *"model construction"* specification

From the merging meta-model, Meeduse automatically generates B specifications that gathers modeling operations as well as structural invariants. This technique allows to write later the transformation rules in the B language. The Meeduse rules for translating an Ecore meta-model into a B specification can be roughly summarized as following:

- Primitive types (*e.g.* integer, boolean) become B basic types ($\mathbb{Z}$, **BOOL**,...).
- For each meta-class there is a variable in the specification (named as the class) representing the set of existing instances (*e.g.* variable *Row* represents the set of existing instances of class `Row`). If a class A is a subclass of a class B then Meeduse generates an inclusion relationship between their corresponding existing instances ($A \subseteq B$). For example, we get predicate $InputPort \subseteq Port$ because class InputPort is a sub-class of class Port. The additional class Element introduced in the merging meta-model is translated into an abstract set that represents all possible instances.
- Each attribute leads to the definition of a variable that is typed as a function from the set of possible instances to the attribute type (*e.g.* $Cell\_value \in Cell \rightarrow BOOL$). The function specializations depend on multiplicities and the optional/mandatory character of the attribute. For example, attribute `selected` in class `Cell` is an optional Boolean attribute, and the corresponding variable is a partial function defined as: $Cell\_selected \in Cell \nrightarrow \mathbf{BOOL}$.
- Associations are represented as functional relations between the sets of possible instances issued from both source and target classes (*e.g.* $tree\_owner \in Tree \rightarrowtail\!\!\!\!\!\!\circ\;\; BDD$). Like attributes, the relation depends on the reference cardinalities (and its opposite) like for example, the *tree_owner* variable which is a partial injection specifying the association between classes Tree and BDD with multiplicity 0..1 in its two extremities.

The behavioral part of the generated B machine provides all basic operations for model manipulation: getters, setters, constructors and destructors; for this reason we refer to this machine as the "model construction"

machine. Note that this step is similar to what happens in MDE tools that generate code from meta-models. For instance, from a given meta-model, EMF generates Java modeling code (getters, setters, etc), that can be used to program model transformation in Java. In the same way, Meeduse generates a B machine that can in turn be used to specify model transformations in B. The B specification generated automatically from the merging meta-model is about 1162 lines of code with 89 basic operations which are proved correct (with respect to the structural invariant) by construction. Proofs were carried out using the theorem prover of AtelierB which generated 260 proof obligations. This means that the use of the modeling operations guarantees the preservation of the structural properties (invariant) of the meta-model and they will never create an invalid instance contrary to a Java-based technique like that of EMF or other tools.

3.3 Step 3: writing and checking the transformation rules

The model transformation is manually written in a new B machine as a set of B operations that call the modeling operations generated in the previous step. Each transformation rule is defined as a B operation composed of two parts: the guard and the action. The guard gives the conditions under which the rule can be triggered, and the action specifies a sequence of calls to modeling operations. Since the individual model construction operations (constructors, setters, ...) were proved correct, the result of executing a sequence of operations in the action part of a rule will obviously preserve the model structural properties. The B specification of the transformation gathers three main B operations (see Appendix for details):

- `TruthTable2BDD`: this rule creates a BDD from a truth table under the condition that the BDD was not previously created. It also creates the BDD input and output ports, and then adds all generated ports to the BDD. It sequentially calls modeling operations `BDD_NEW`, `BddInput_NEW`, `BddOutput_NEW` and `BDD_Addports` which were generated from the meta-model.

- `SelectPort`: this rule selects an input port satisfying a maximality condition that depends on the current state of the transformation and then decides whether it creates a new tree or reuses a tree already created. When it creates a tree it calls the modeling operation `Subtree_NEW`. For the first tree it only calls `Tree_SetOwnerBDD` which marks this first tree as a root tree. These are the first actions that the operation makes. The next actions non-deterministically select cells of value zero or one which leads to two possible instances of operation `SelectPort` that can be applied to the same selected port.

- `Transform`: this operation can be triggered only when there is no more than one selected row, and allows to consume the row together with its cells. It has two deterministic behaviours: creates an assignment for output cells if there exists an output cell not yet consumed, or creates a leaf if all output cells are consumed.

In order to verify the correctness of our rules we introduce invariants that define the transformation properties and we apply a model-checking proof in order to check for the existence of a sequence leading to violations. Since we deal with a bounded state-space, this proof is sufficient as far as the state-space is entirely covered. The ProB model-checker computes exhaustively all the execution possibilities and checks the reachability of unwanted states using the following goals:

- For every consumed row, one distinct leaf is created.

- For every output cell of a consumed row, one assignment is created with the same value.

- When there is no row to deal with then all tree links are produced.

- Values of trees in a computed BDD path are equivalent to the selected cells values in the consumed row.

Our exhaustive model-checking validation technique was done on input models of reasonable sizes: until 5 input ports, 2 output ports, and 32 rows. We believe that the model-checking proofs done given these models are sufficient to have confidence about our rules because most of the provided models are generated by a combinatorial technique. We think that since the proof succeeded for a restricted number of port combinations, then it can be generalized for bigger combinations. Not only the algorithm applies redundantly the same principles to the consumed rows but also the properties of these rows (by means of cell values and connexion with ports) are similar and they don't change during the transformation.

Further study may be required in order to show the existence of a least fixed point, from which one can generalize the proof and stop building input models for the exhaustive model-checking. For bigger examples we simply apply Meeduse as a runner of the transformation in order to get the output BDD. For these examples we set property `SET_PREF_MAX_OPERATIONS` to one, which forces ProB to compute only one instance of each operation which is immediately animated by Meeduse in the automatic execution mode. Finally, we note that all our output models successfully passed the validator provided by the TTC'2019 organizers which was somehow expected since we checked the B specifications using automated reasoning tools. Meeduse was also helpful for debugging the formal specifications thanks to the visualization designed in Sirius.

4 Execution

Proofs are mainly for verification purposes (i.e., *"is the transformation correct?"*). However, we need to validate the rules in order to be sure that they produce the results expected by a domain expert (i.e., *"is this the right transformation?"*). For this purpose, Meeduse provides an interactive animation facility that uses the ProB [5] animator in the background. When executed on a given root element of an EMF resource, Meeduse is synchronised with the resource and every Eclipse tool also synchronised with the same resource is expected to be compatible with Meeduse.

In our solution, one can use our Sirius artifacts for visualizing the models (the TT and the BDD) issued from the merging meta-model. Sirius has two benefits: (1) it favours graphical animation because when executed the model changes (input elements are consumed and output elements are produced) and Sirius automatically updates its rendering at every modification of the model, and (2) it is an easy way to define conditional styles which changes the visual representation depending on some OCL-like conditions. For example, when a cell is selected it becomes green which allows the user to know which rows are being transformed. Figure 2 shows the Sirius views of a truth table under transformation and the current state of the corresponding BDD. In this snapshot some cells of rows `r_6` and `r_7` are currently selected (for ports `a, c, d`), and a sub-part of the BDD was produced from rows already consumed (`r_5` and `r_8`). In our transformation a consumed row is simply removed from the truth table.

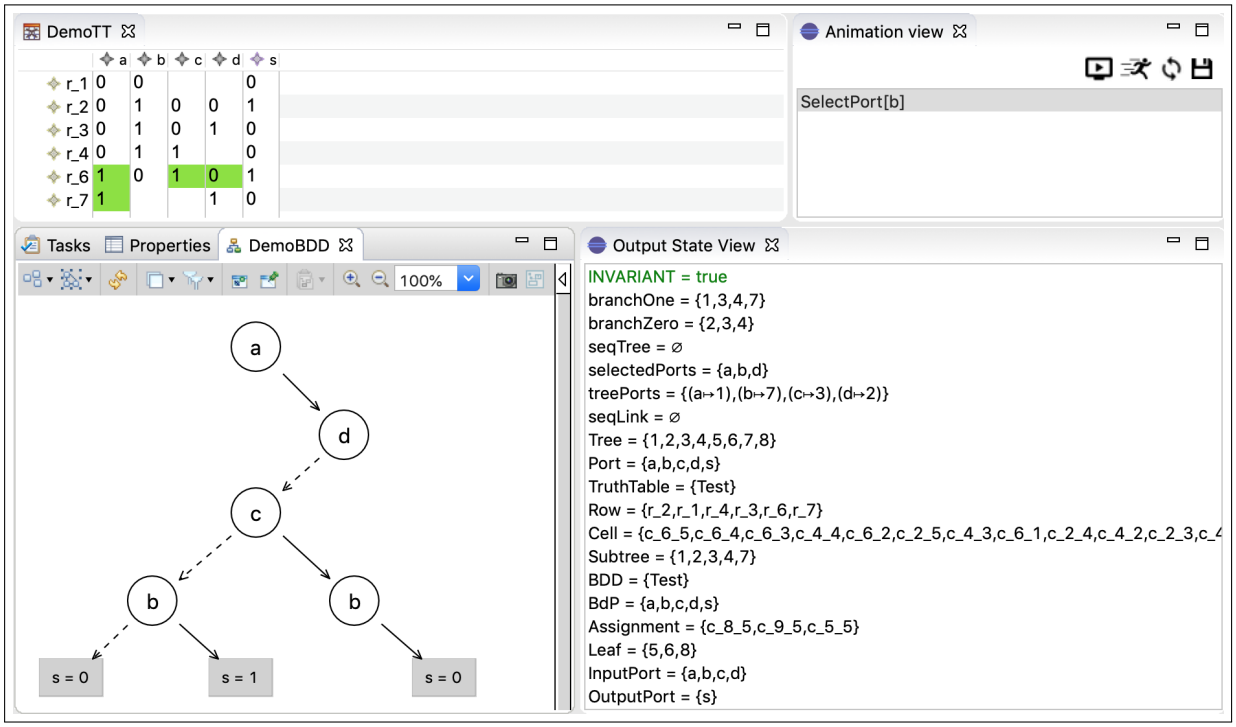


Figure 2: A Truth Table under transformation

5 Discussion and conclusion

This work used together four tools:

- EMF for meta-modeling and the automatic extraction of an editor plugin in a classical MDE approach.
- The AtelierB prover for theorem proving in order to prove that the various B specifications (the model construction and the model transformation) preserve the meta-model structural properties.
- ProB for one of its numerous model-checking capabilities.
- Meeduse was involved in order to translate an Ecore meta-model into a B specification, and also for debugging and executing the transformation given instance of the input meta-model.

In our approach, we are not advising that the MDE expert learns the B method and its associated tooling, or conversely that the FM expert learns meta-modeling with its tools. We believe that skills in both domains are required, and we suggest a way to make people collaborate. Meeduse provides practical solutions for that, as shown by this application. Furthermore, we exploited neither the whole MDE capabilities nor the whole FM capabilities, but we limited our proposal there to a subset of what can be done for the particular case of model-to-model transformations. From a methodological point of view we were able to define how formal DSL execution semantics can be applied to define model transformations. In general, we are satisfied with the application of Meeduse to the model-to-model transformation problem because as far as we know none of the existing works combine theorem proving and model-checking in a publicly available tool which is well-integrated within EMF-based platforms. The performance mainly depends on the performance of ProB. When the number of model elements grows exponentially (14 input ports and 4 output ports) Meeduse went out of memory. For bigger examples, it should be useful to try the experiments on a machine with higher performances than that on which we have done these measures.

For readability, we believe that the verbose notation of the B method is accessible (refer to the appendix) because it recalls some programmatic styles. It is often said to be less difficult than other formal notations. Our transformation file is about 150 lines which remains reasonable. We think that model transformation interests the safety-critical community whose main intention is to develop systems which are bug-free because a failure can lead to human loss. This study gives solutions to this problem with the support of a tool and advocates for a collaboration between MDE and FM experts.

References

[1] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] Antonio Garcia-Dominguez and Georg Hinkel. Truth Tables to Binary Decision Diagrams. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 12th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2019) federation of conferences*, CEUR Workshop Proceedings. CEUR-WS.org, July 2019.

[3] A. Idani, Y. Ledru, A. Ait Wakrime, R. Ben Ayed, and P. Bon. Towards a tool-based domain specific approach for railway systems modeling and validation. In *Third International Conference, RSSRail*, volume 11495 of *LNCS*, pages 23–40. Springer, 2019.

[4] A. Idani, Y. Ledru, A. Ait Wakrime, R. Ben Ayed, and S. Collart Dutilleul. Incremental Development of a Safety Critical System Combining formal Methods and DSMLs. In *24th International Conference on Formal Methods for Industrial Critical System (FMICS)*, volume 11687 of *LNCS*, pages 93–109. Springer, 2019.

[5] Michael Leuschel and Michael Butler. ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, Mar 2008.

Appendix

REFINEMENT
meeduse_tt2bddref
REFINES
meeduse_tt2bddmain
INCLUDES
meeduse_tt2bdd
DEFINITIONS
$selectedRows ==$
$\quad$ **LET** cr **BE** $cr = \{cc,rr \mid rr \in Row \wedge cc = \mathbf{card}(cells^{-1}[\{rr\}] \cap selectedCells)\}$ **IN**
$\quad\quad$ **LET** mx **BE** $mx = \mathbf{max}(\mathbf{dom}(cr))$ **IN**
$\quad\quad\quad \mathbf{cr}[\{mx\}]$
$\quad\quad$ **END**
$\quad$ **END**;

$portRow(rr) == (cellPort^{-1} ; cells) \rhd rr$;

$maxPort(pp,rr) == pp \in InputPort \wedge rr \subseteq Row \wedge$
$\quad \neg (\exists ss . (ss \in InputPort \wedge ss \neq pp \wedge ss \in \mathbf{dom}(portRow(rr))$
$\quad\quad \wedge \mathbf{card}(portRow(rr)[\{ss\}]) > \mathbf{card}(portRow(rr)[\{pp\}]))$;

$zeroCells(pp) == (cellPort^{-1}[\{pp\}] \cap cells^{-1}[selectedRows]) \cap Cell\_value^{-1}[\{\mathbf{FALSE}\}]$;

$oneCells(pp) == (cellPort^{-1}[\{pp\}] \cap cells^{-1}[selectedRows]) \cap Cell\_value^{-1}[\{\mathbf{TRUE}\}]$;

$selectedCells == \mathbf{dom}(Cell\_selected \rhd \{\mathbf{TRUE}\})$;

$outputCells(rr) == cells^{-1}[\{rr\}] \cap cellPort^{-1}[OutputPort]$;

$inputCells(rr) == cells^{-1}[\{rr\}] \cap cellPort^{-1}[InputPort]$

VARIABLES
branchOne, *branchZero*,
seqTree, *selectedPorts*, *treePorts*, *seqLink*
INVARIANT
$branchOne \subseteq Tree \wedge$
$branchZero \subseteq Tree \wedge$
$selectedPorts \subseteq Port \wedge$
$treePorts \in InputPort \leftrightarrow Tree \wedge$
$seqTree \in \mathbf{seq}(Tree) \wedge$
$seqLink \in \mathbf{seq}(\mathbf{BOOL})$
INITIALISATION
$branchOne, branchZero, selectedPorts := \emptyset , \emptyset , \emptyset \,||$
$treePorts, seqTree, seqLink := \emptyset , \emptyset , \emptyset \,||$
$setLastTree(\mathbf{card}(Subtree))$

OPERATIONS
TruthTable2BDD =
  ANY *src* **WHERE**
    $src \in TruthTable \wedge src \notin BDD$
  THEN
    BDD_NEW(*src*) ;
    BddInput_NEW(*InputPort*) ;
    BddOutput_NEW(*OutputPort*) ;
    BDD_Addports(*bdd*, $InputPort \cup OutputPort$)
  END;

SelectPort =
  ANY *port* **WHERE**
    $InputPort \neq \emptyset$
    $\wedge\ port \in BddInput$
    $\wedge\ port \notin cellPort[selectedCells]$
    $\wedge\ maxPort(port, selectedRows)$
    $\wedge\ \mathbf{ran}(seqTree) \cap Leaf = \emptyset$
  THEN
    SELECT
      $port \in selectedPorts$
    THEN
      $seqTree := seqTree \leftarrow (treePorts(port))$
    WHEN
      $port \notin selectedPorts$
      $\wedge \neg (\exists\ portBis\ .\ (portBis \notin cellPort[selectedCells]$
          $\wedge\ maxPort(portBis, selectedRows)$
          $\wedge\ portBis \in selectedPorts))$
    THEN
      Subtree_NEW(*port*) ;
      BEGIN
        $selectedPorts := selectedPorts \cup \{port\}$ ||
        treePorts(*port*) := *lastTree* ||
        $seqTree := seqTree \leftarrow (lastTree)$
      END ;
      IF $lastTree = 1$ **THEN**
        Tree_SetOwnerBDD(*lastTree*, *bddPorts*(*port*))
      END
    END ;

    SELECT $zeroCells(port) \neq \emptyset$ **THEN**
      Cells_SetSelected(*zeroCells*(*port*), **TRUE**) ||
      $branchZero := branchZero \cup treePorts[\{port\}]$ ||
      $seqLink := seqLink \leftarrow$ (**FALSE**)
    WHEN $oneCells(port) \neq \emptyset$ **THEN**
      Cells_SetSelected(*oneCells*(*port*), **TRUE**) ||
      $branchOne := branchOne \cup treePorts[\{port\}]$ ||
      $seqLink := seqLink \leftarrow$ (**TRUE**)
    END
END;

setLinks =
  ANY *t1*, *t2* **WHERE**
    $t1 = \mathbf{first}(seqTree) \wedge t2 = \mathbf{first}(\mathbf{tail}(seqTree))$
    $\wedge\ \mathbf{ran}(seqTree) \cap Leaf \neq \emptyset$
    $\wedge\ \mathbf{card}(seqTree) > 1$
  THEN
    IF $\mathbf{first}(seqLink) = \mathbf{TRUE}$ **THEN**
      Subtree_SetTreeForOne(*t1*, *t2*) ||
      $seqLink := \mathbf{tail}(seqLink)$
    ELSE
      Subtree_SetTreeForZero(*t1*, *t2*) ||
      $seqLink := \mathbf{tail}(seqLink)$
    END ||
    $seqTree := \mathbf{tail}(seqTree)$
  END;

Continue =
  SELECT
    $\mathbf{card}(seqTree) = 1 \wedge \mathbf{ran}(seqTree) \cap Leaf \neq \emptyset$
  THEN
    $seqTree := \mathbf{tail}(seqTree)$
  END ;

Transform =
  ANY *row* **WHERE**
    $row \in selectedRows$
    $\wedge\ \mathbf{card}(selectedRows) = 1$
    $\wedge\ \forall\ cc\ .\ (cc \in cells^{-1}\ [\{row\}] \wedge cellPort(cc) \notin OutputPort \Rightarrow Cell\_selected(cc) = \mathbf{TRUE})$
  THEN
    IF $\mathbf{card}(outputCells(row)) > \mathbf{card}(assignPort[outputCells(row)])$ **THEN**
      ANY *as* **WHERE** $as \in outputCells(row) \wedge as \notin Assignment$ **THEN**
        Assignment_NEW(*as*, *cellPort*(*as*), *Cell_value*(*as*))
      END
    ELSE
      Leaf_NEW ;
      $seqTree := seqTree \leftarrow (lastTree)$;
      Assignments_SetOwner(*outputCells*(*row*), *lastTree*) ;
      Cells_Free($inputCells(row) \cup outputCells(row)$) ;
      $selectedPorts := selectedPorts$ -
          $\{app \mid app \in selectedPorts \wedge treePorts(app) :\ (branchZero \cap branchOne)\}$;
      Row_Free(*row*)
    END
  END
END

Description of the transformation rules

The definition clause allows to define some kinds of helpers (we reuse the ATL term) that calculate some formula based on the set theory and the first order logic predicates:

$$zeroCells(pp) == (cellPort^{-1}[\{pp\}] \cap cells^{-1}[selectedRows]) \cap Cell\_value^{-1}[\{\mathbf{FALSE}\}] ;$$

$$oneCells(pp) == (cellPort^{-1}[\{pp\}] \cap cells^{-1}[selectedRows]) \cap Cell\_value^{-1}[\{\mathbf{TRUE}\}] ;$$

$$selectedCells == \mathbf{dom}(Cell\_selected \rhd \{\mathbf{TRUE}\}) ;$$

$$outputCells(rr) == cells^{-1}[\{rr\}] \cap cellPort^{-1}[OutputPort] ;$$

$$inputCells(rr) == cells^{-1}[\{rr\}] \cap cellPort^{-1}[InputPort]$$

- *zeroCells* applied to a port *pp* gives all cells with value 0 that belong to the selected rows. The row selection mechanism will be discussed while presenting the transformation rules. Definition *oneCells* is similar but gives cells with value 1.
- *outputCells* and *inputCells* applied to a row *rr* gives the cells that are concerned by an output or an input port.
- *selectedCells* gives the set of cells that are consumed during the transformation.

The algorithm proposed in the TTC'2019 call for solutions suggests to find an input port which is (ideally) defined in all the Rows, and turn it into an inner node. We somehow applied this technique but introduced a maximality criterion. In fact our algorithm chooses the port whose set of cells is the biggest one, with respect to the selected rows.

A step-by-by step execution

The screen-shot of figure 3 shows that at the beginning of the transformation the only port that can be selected is port `a`. This is the expected result since port `a` establishes the maximality criterion. Note that in the initial state all rows are selected and then `a` has the biggest set of cells in comparison with the other ports.

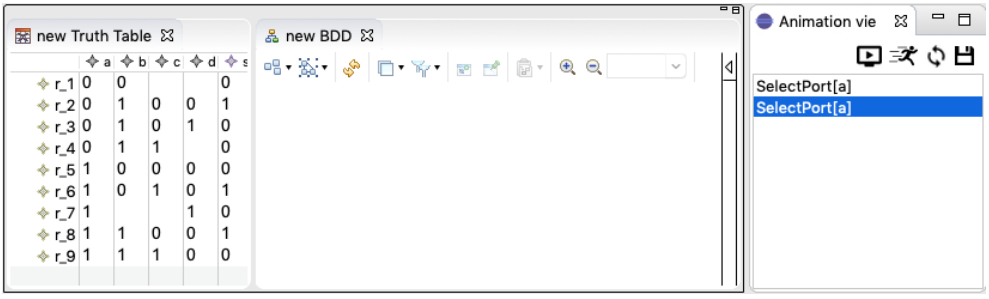


Figure 3: First execution

The `Animation view` provides two possibilities for `selectPort(a)` because one can select the zero value or the one value. The animation of the second occurrence of `selectPort(a)` leads to figure 4 where cells of value one of port `a` are selected and a node is created in the BDD model. In fact, every time a port is selected, a node in the BDD is created. For this state, formula *maxPort* identifies port `d` as the one satisfying the maximality criterion and then the animation view gives two possible executions of `selectPort(d)` (for value zero and for value one).

The animation of the first occurrence of `selectPort(d)` leads to the model of figure 5 where zero cells of port `d` are selected and an other node is created in the BDD model. In this new state four possible rules can be triggered because ports `b` and `c` are equivalent regarding the maximality criterion. Meeduse suggests then two

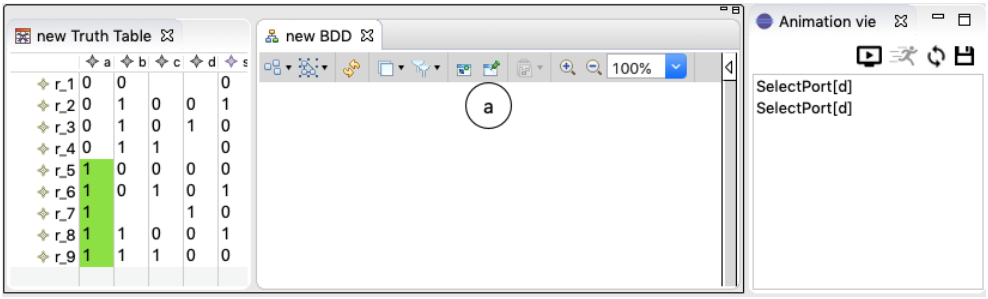


Figure 4: Second execution

possibilities for each of `selectPort(b)` and `selectPort(c)`. Running the second occurrence of `selectPort(b)` produces figure 6 from which it is possible to trigger finally rule `selectPort(c)` and hence reach the end of the selection step with nodes extraction.

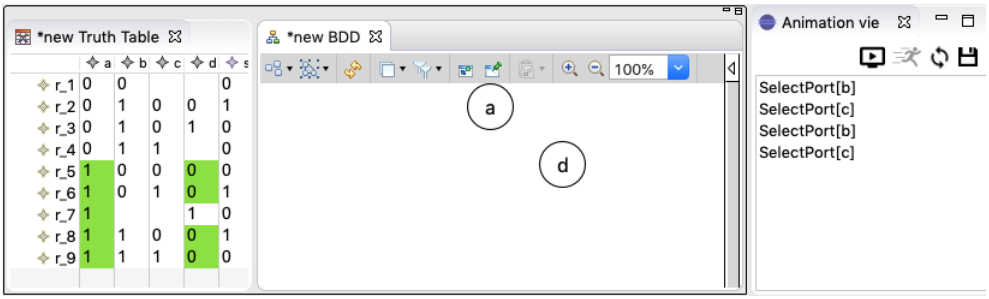


Figure 5: Third execution

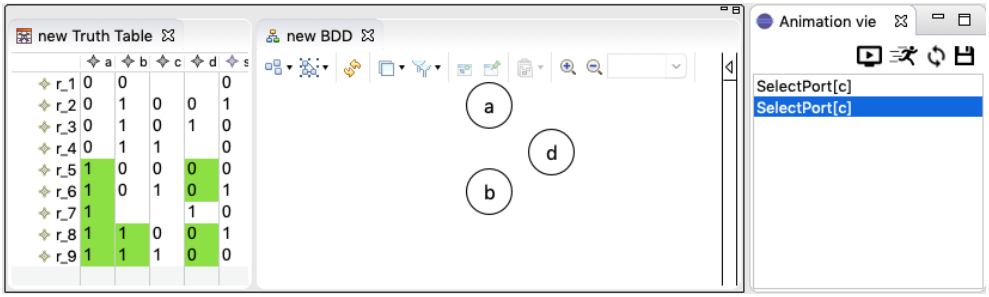


Figure 6: Fourth execution

From step of figure 6 the execution of the second occurrence of `selectPort(c)` leads to figure 7 where only one row `r_9` has all its cells selected. Now, only rule `Transform(r_9)` is proposed. When applied this rule iterates several times on row `r_9` until it transforms it entirely. The first calls transform non-deterministically the row output cells into assignments with the same values (figure 8). After consuming all output cells (in this case we have only one output cell), this rule creates a Leaf and then removes the row from the model together with its cells. By this way row `r_9` and its cells will not be considered for the next calculus of the enabledness conditions of the transformation rules.

In figure 9, after removing row `r_9`, the enabled rules are those that create links between nodes, assignments and leafs. These are successive occurrences of operation `setLinks`: `setLinks(1,2)`, `setLinks(2,3)`, `setLinks(3,4)`, `setLinks(4,5)`. The valuations correspond to tree identifiers managed by the internal state of the B specification every time an instance of class tree is produced.

Figure 10 gives the resulting model after a row is entirely consumed and the corresponding path in the BDD is produced. Rule `continue` then updates the internal state of the B machine and makes possible the port selection process for the remaining rows. From the model of figure 10 only port `c` with value zero can be selected. Indeed, given the set of selected rows and cells, only port `c` satisfies the maximality criterion.

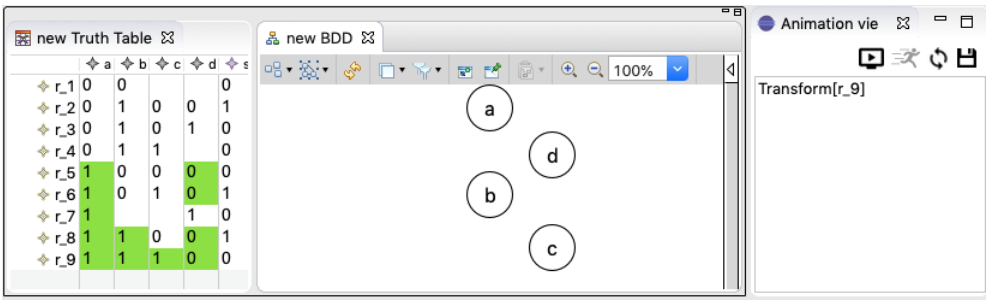


Figure 7: Fifth execution

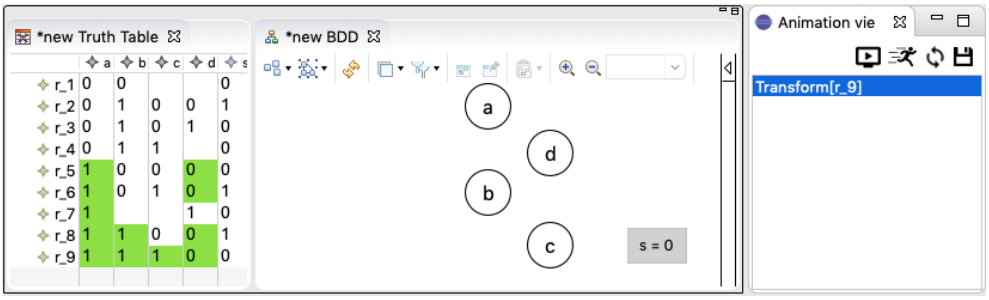


Figure 8: Sixth execution

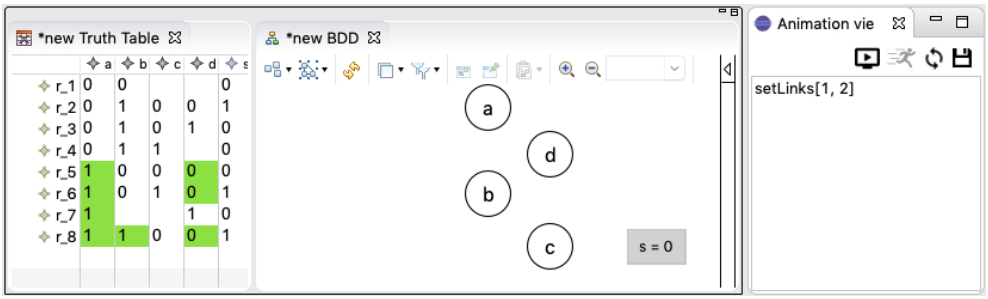


Figure 9: Seventh execution

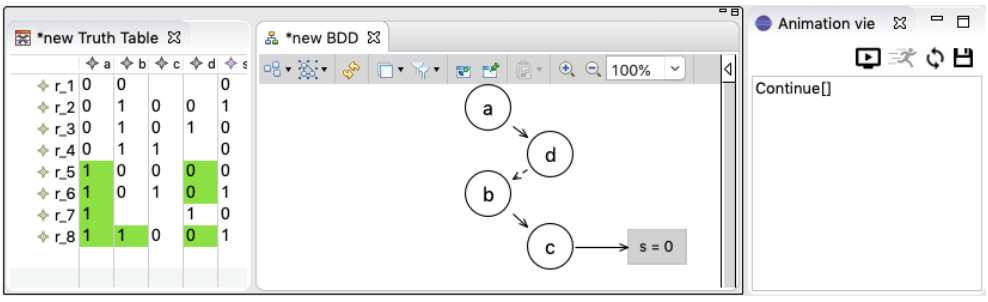


Figure 10: A successive animation of setLink

Transforming Truth Tables to Binary Decision Diagrams using the Role-based Synchronization Approach

Christopher Werner, Rico Bergmann, Johannes Mey, René Schöne and Uwe Aßmann
{christopher.werner,rico.bergmann1,johannes.mey,rene.schoene,uwe.assmann}
@tu-dresden.de

Software Technology Group
Technische Universität Dresden

Abstract

The Transformation Tool Contest (TTC) 2019 case describes the computation of a binary decision tree (BDT) or diagram (BDD) from a given truth table. This paper presents a complete solution of the case with the Role-based SYNChronization approach (RSYNC) that is based on the role concept. The solution contains detailed transformation algorithms that create an ordered and unordered BDT and BDD which differ in the port orders for each subtree. The transformation algorithms run after instantiating the truth table. In addition, a solution is presented that creates the BDT directly at instantiation phase of the truth table. We evaluate our RSYNC transformation approach and show the advantages of the role concept in such a transformation.

1 Introduction

Every year, the Transformation Tool Contest (TTC) chooses a task for the evaluation of different transformation approaches. This year, the TTC task is to transform a truth table (TT) into a binary decision diagram. The presented case [GDH19] provides two different target metamodels to choose from. These metamodels differ in the representation as a tree (BDT) and graph structure (BDD). To validate the overall solution, the target models must correspond to the source model. Besides that, there are no requirements regarding the order of the nodes or the optimality of the target model. To solve this problem, we use the Role-based SYNChronization (RSYNC) [WSK+18] approach, which describes rules for creating, deleting, modifying, and transforming elements from the source to the target model. All rules in the approach are modelled as roles and compartments with the role concept and automatically set up explicit traceability links with the *plays* relations between the source and target models. These traceability links allow incremental change propagation between the models which means that the source and target models can be kept consistent at runtime with a consistency preserving mechanisms. We present the usability of the RSYNC approach for the TTC case and compare the RSYNC approach with the other ones. The implementation of the TTC case with RSYNC can be found in the TTC GitHub repository[1] and in our public Git repository[2].

The next section summarizes background knowledge about closely related topics. Section 3 provides the abstract transformation algorithm that is used to create a BDT and a BDD from the TT. Section 4 describes the overall transformation chain, which must be performed to create the target model. We compare our approach to the current ones in Section 5. Finally, in Section 6, we conclude the paper and discuss lines of future work.

[1] https://github.com/TransformationToolContest/ttc2019-tt2bdd
[2] https://git-st.inf.tu-dresden.de/ttc/bdd

In: A. Garcia-Dominguez, G. Hinkel and F. Krikava (eds.): Proceedings of the 12th Transformation Tool Contest, Eindhoven, The Netherlands, 19-07-2019, published at http://ceur-ws.org

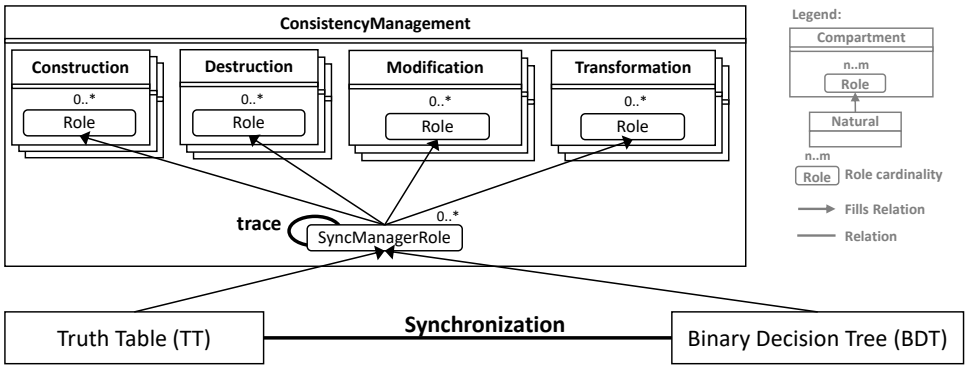


Figure 1: Role-based synchronization approach (RSYNC).

2 Background

The role-based synchronization approach (RSYNC) [WSK+18] used in this paper is based on the role concept known from the 1970s [BD77]. In the 2000s, Steinmann and Kühn *et al.* [Ste00, KLG+14] identified 27 features that describe the nature of roles in terms of their behavior, their relational dependence to each other, and their context dependency. These properties were depicted using the Compartment Role Object Model (CROM), where instances can be modelled with FRaMED [KBRA16]. The role concept offers an extension of the object-oriented paradigm and is suitable for processes that change over time because of simple runtime adaptation and evolution mechanism. Transformations usually describe a one-time change step but must be adapted and executed again if the source model or target model changes. For this reason, the role concept is suitable in the area of model transformation and synchronization for permanent consistency preservation of several models.

The RSYNC approach uses the advantages of roles and describes an approach for synchronizing multiple related models at runtime. In Figure 1, the concept is visualized on metamodel level with two blocks representing the TT and BDT metamodels and the synchronization between them. The core of the approach is the `ConsistencyManagement` compartment, which manages all rules of the synchronization and takes care of the execution. In the role concept, compartments represent a context in which roles exist and interact with each other. The RSYNC approach distinguishes four types of rules, each of which is modelled in their own compartments: (1) *Construction rules* describe what happens when new model elements are created in one of the connected models and trigger the creation of elements in the other models. (2) *Destruction rules* describe how to deal with the deletion of elements. (3) *Modification rules* indicate how to deal with attribute or reference changes in connected models. (4) And *transformation rules* describe rules on how to create a completely new instance model from an existing one. The transformation rules describe the creation of a new BDD or BDT model from an existing TT model and are sufficient enough for the TTC case. Most approaches only allow the transformation between several models but do not support the runtime consistency management. For an incremental synchronization at runtime, the other rule types must be implemented and are discussed in the following sections.

The RSYNC approach is implemented in the SCala ROle Language (SCROLL) [LA15] which supports most of the 27 role features. The implementation allows the exchange of rules and the integration of new models at runtime. Each model element can play roles in the rule compartment of the different types and will get informed about changes in the other models without knowing the concrete connection. In addition, the *plays* relationships of the role concept allow the explicit description of traceability links between the different models.

3 Computing a Binary Decision Tree and Diagrams with RSYNC

This section describes the algorithm for generating a BDT and BDD from a TT. It uses one algorithm with a predefined port order and one generating a lower number of decision nodes in different order heuristically. Furthermore, we show these algorithms as simple transformations in the RSYNC approach and a transformation running at runtime in the initialization phase that reacts on changes and synchronizes the source and target models directly.

3.1 Transform a Truth Table to a Binary Decision Tree

The RSYNC approach uses the already presented algorithm from the task paper [GDH19]. A `BDD` object is created for each `TruthTable` object and all `Ports` in the input model create a corresponding `Port` in the output

```
1 class TruthTableTransformation() extends ITransformationRole {
2   def transform(comp: PlayerSync): PlayerSync = {
3     val name: String = +this getName ()                          //get name from source object
4     val ports: Set[tt.Port] = +this getPorts ()                  //get ports from souce object
5     val bdd = new bdd.BDD(name,null,Set.empty)                   //create object in target model
6     connectTargetElementWithSourceElement(bdd,comp)              //add traceability information
7     ports.foreach(p => {
8       val subRule = getSubTransformation(p)                      //get subrule for ports
9       val manager: ISyncManagerRole = +p getManager ()           //get own SyncManager role
10      if (manager != null) {
11        manager play subRule                                     //add subrule
12        val o = subRule.transform(p).asInstanceOf[bdd.Port]      //run subrule
13        subRule.remove()                                         //remove subrule
14        o.setOwner(bdd)                                          //create relations
15        bdd.addPorts(o)
16      }})                    /* more sub transformation rules can run here */
17    return bdd
18  }
19 }
```

Listing 1: Transformation rule from a `TruthTable` object to a `BDD` object.

model. Then, the subtrees for the graph are created, whereby the port order is determined in two ways: (1) using a fixed order from the beginning (ordered), or (2) the port for the following subtree is selected after splitting the rows, whereby the leaves with their assignments are optimally divided. When there are more or no optimal splits, the port with the most rows or the first one found is used (unordered). Afterwards, the instances of the target models are created using one of these two variants. The difference between BDT and BDD as target model lies in the merging of the same leaf nodes in the BDD to generate the minimum number of leaves.

3.2 Transformation Rules in the RSYNC Approach

For the transformation from the TT Model to the BDD Model, it is necessary to implement a transformation compartment, which contains roles as implementation of the transformation rules and subrules. The transformation rule to transform a `TruthTable` object into a `BDD` object is illustrated in Listing 1. The role `TruthTableTransformation` contains the *transform* method, which is called when a new `TruthTable` object is found in the source model. Transformation rules always return the newly created object which is of type `PlayerSync` because each element must extend from this type to be integrated in the RSYNC environment. This rule represents the entry point for the global transformation and then calls subrules (e.g., lines 11-13 for all ports). In lines 3 and 4, the data of the source element is retrieved; it is important that this role is played by the source element, which is achieved by the RSYNC approach. The + operator allows to call methods that are implemented in other roles or the player itself. In line 5, the object of the target model is created and in line 6, the objects are linked together, integrated into the framework, and get modification rules if necessary. Listing 1 shows only a part of one transformation compartment. All transformation compartments can be found in the *ttc2019.sync* package. Currently these transformation rules have to be implemented by hand in Scala, but in the future an existing transformation language shall be used or a new simple language should be developed, which generates the skeletons of the rules, where the concrete implementation can be made by hand.

3.3 Synchronization of a Truth Table and a Binary Decision Tree

The previous section describes the transformation from a source to a target model with several transformation rules. The main benefit of RSYNC is the possibility to offer construction, destruction, and modification rules to create a target model directly while instantiating the source model and to propagate changes between both models at runtime. To demonstrate this, we implemented rules based on the standard algorithm to directly generate the BDT when creating a TT. There are construction rules that are called when a `TruthTable` object is created to automatically create a corresponding `BDD` object and similar rules for ports. Since these implementations are similar to those in the previous section, we do not directly address their implementation here. There are also modification rules (methods that are bound to objects by roles) that keep the names of `Ports` and `TruthTable` objects in sync with their counterparts (`SyncPortNames`). In addition, we implemented modification rules that are executed when a port or line is added to a `TruthTable` object.

```
1 def syncAddPorts(port: PlayerSync): Unit = {
2   val oBdtBDD: PlayerSync = +this getRelatedObject ("bdd.BDD")    //get connected BDD from tree
3   val oBdtPort: PlayerSync = +port getRelatedObject ("bdd.Port")  //get connected Port from tree
4   if (oBdtBDD != null && oBdtPort != null) {                      //check existence
5     val bdtBDD = oBdtBDD.asInstanceOf[bdd.BDD]                    //cast elements
6     val bdtPort = oBdtPort.asInstanceOf[bdd.Port]
7     bdtBDD.addPorts(bdtPort)                                      //make connection
8     bdtPort.setOwner(bdtBDD)
9   }
10  val oBddBDD: PlayerSync = +this getRelatedObject ("bddg.BDD")   //get connected BDD from tree
11  val oBddPort: PlayerSync = +port getRelatedObject ("bddg.Port") //get connected Port from tree
12  if (oBddBDD != null && oBddPort != null) {                      //check existence
13    val bddBDD = oBddBDD.asInstanceOf[bddg.BDD]                   //cast elements
14    val bddPort = oBddPort.asInstanceOf[bddg.Port]
15    bddBDD.addPorts(bddPort)                                      //make connection
16    bddPort.setOwner(bddBDD)
17  }
18 }
```

Listing 2: Modification rule reacts on adding ports for the synchronization example.

Listing 2 shows the modification rule for adding a `Port` to a `TruthTable` which is automatically executed when the *addPort* method in the `TruthTable` class is called. Since connected objects in the target model are already created from the `Port` and `TruthTable` objects of the source model with the construction rules, the connected elements are found in the target model via the traceability links (lines 2 and 3). After an existence check (line 4), the new elements get connected in the target model (lines 5-8). In addition, the steps in lines 10-17 can also be executed for a second target model. These rules show the fifth transformation of the TTC case, whereby this step runs completely in the initialization phase and does not require a transformation phase.

4 The Transformation Toolchain

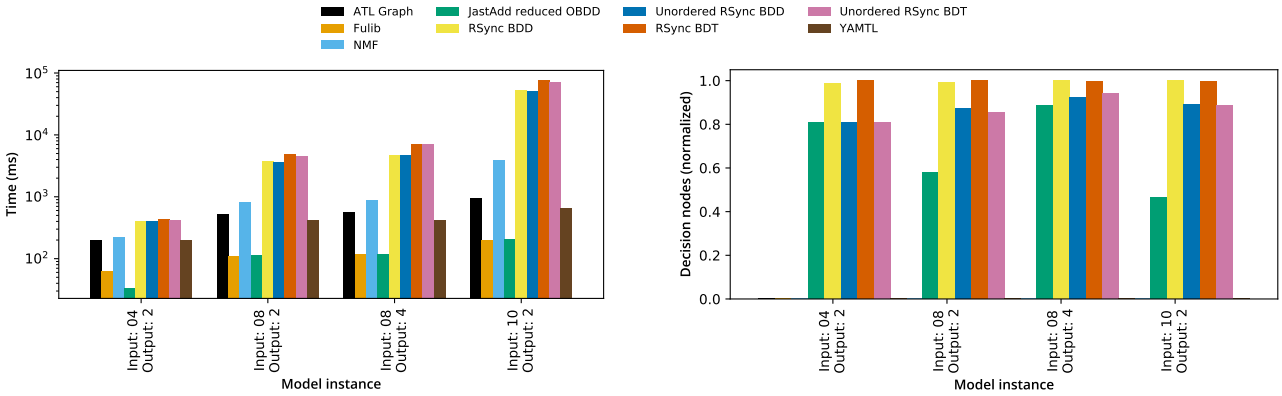
This section describes the necessary process steps to implement the transformation with the RSYNC approach. The complete approach is implemented in Scala and is based on the role-based programming language SCROLL [LA15]. For this reason, the source and target models must be available as Scala classes in order to create a system that keeps them consistent at runtime. For this step a code generator is available, which generates Scala classes from an Ecore model. The classes represent the complete model and get additional information for the integration into the RSYNC approach. In addition, the generator creates classes and methods to read the XMI instance models of the metamodel. In the next step, the generated classes are integrated into the RSYNC environment and form the basis for the synchronization and transformation. After generating and integrating the classes, the rules have to be implemented. Currently, there is no DSL support for the rule implementation which should be added in the future. The rules are implemented as classes that inherit from special rule interfaces in order to be directly integrated into the RSYNC system. Listing 3 shows the adding of all kinds of rules to the `ConsistencyManagement` compartment. It contains all rules for a transformation and synchronization approach. In line 1, a transformation rule as shown in Listing 1 is added to the `ConsistencyManagement` compartment. If transformation rules are added to the `ConsistencyManagement` compartment, they will be directly executed when possible matches exist. Lines 2-5 show the adding of all necessary rules for the synchronization of the models. A part of the `SyncTruthTableModifications` rule was already presented in Listing 2. Such rules must be added before instantiating the source model, so that instances in the target model are created during instantiation phase. After the transformation, the target models are saved as XMI models. Since this step is not part of the RSYNC environment, it is implemented manually using the Java classes from the TTC repository. The validation of the output model is also implemented with the existing Java classes of the repository and is always checked after the transformation.

5 Evaluation

The evaluation of the presented approach is presented in two parts. First, we examine different non-functional properties of our solution and afterwards the benchmark results are presented and discussed.

```
ConsistencyManagement.transformModel(BdtTransformation)                //add transformation rule
ConsistencyManagement.changeConstructionRule(TtBdtBddConstruction)     //add construction rule
ConsistencyManagement.addModificationRule(new SyncPortNames)           //add modification rules
ConsistencyManagement.addModificationRule(new SyncTruthTableModifications)
ConsistencyManagement.addModificationRule(new SyncCellModifications)
```

Listing 3: Add all kinds of rules to the `ConsistencyManagement` compartment.



(a) Total time.

(b) Number of decision nodes (normalized).

Figure 2: Evaluation result.

5.1 Properties of the Transformation

By applying our role-based programming infrastructure and supported tools, the provided ecore models can be reused directly. Our code generator framework creates the necessary code to represent the ecore models in Scala and to import the XMI models directly. Thus, close to no boilerplate code except the generated code is necessary. Furthermore, due to the unidirectional transformation, the necessary transformation rules have to be implemented as new transformation compartments. As mentioned earlier this is currently done manually but will be subject to automation in the future.

The transformation compartment adheres to an abstract interface which enables the simple adaptation for the transformation process. Different strategies might be used by simply swapping the concrete implementation of the compartment (or parts of it). The approach is utilized for creating the BDTs or BDDs in an ordered and unordered variant. In addition, we implemented different modification rules that make the transformation directly at initialization time without the need of implemented transformation rules.

5.2 Benchmark results

To evaluate the runtime performance, we present the measurements of the TTC organizers. They were performed on a Google Cloud Compute Engine c1-standard-4 machine with 16GiB of RAM and 30GB SSD, with a Docker image[3]. Each configuration of a tool was run ten times. In addition to the suggested time measurement, we measure the number of decision nodes and the minimum, maximum, and average path length of the resulting binary decision diagram. These metrics provide an overview of the memory consumption of the output model and the average iteration time of the diagram. We chose not to measure memory consumption as it heavily depends on details of the target platform (such as the internals of the JVM memory organization). Therefore, a better indication for memory usage would be the number of additional allocated objects (that is others than the objects of source and target models). Here, we currently only need one extra object (`SyncManagerRole`) for each object which acts as the traceability and management handler role. When the use case becomes more dynamic, i.e. truth tables are being created or modified during runtime, this number will increase in minimal manner.

The benchmark results are presented in Figure 2. The load time of the TT model will increase to a minimal extent which is due to the increase of its size. In addition, the results show that most of the time is spent creating the objects and roles with their role bindings. The RSYNC approach is therefore the slowest compared to all other measured approaches but there is a lot of potential for optimizing our approach and the underlying

[3] `https://github.com/TransformationToolContest/ttc2019-tt2bdd/blob/master/Dockerfile`

role-based programming language SCROLL. The *"Input 12 Output 2"* model (as well as the more complex ones) may not be transformed within the given time boundaries. Furthermore, the BDT approaches take longer to be created than the BDD ones which is due to the continuous allocation of new leaf nodes.

Finally, the influence of our heuristic is visible in the total time and the size of the resulting decision structure. In both cases, the heuristic leads to slightly better results. The number of decision nodes is reduced by approximately 10% with a 20% reduction for each model. The results of the average path length show that it is always about half a node below the maximum path length, i.e., the heuristic does not have a large influence on the iteration time. With more optimization steps after the transformation, it will be possible to reduce the number of decision nodes and the average path length again as shown in the JastAdd solution but such a step is not implemented here. As the initialization time is constant in our case, we do not consider it any further here.

6 Conclusion and Future Work

We have shown how to apply the role-based synchronization (RSYNC) approach to the problem of transforming TTs to BDTs and BDDs. In order to do so, we utilized SCROLL as a role-oriented extension of the Scala programming language and applied its features to build an infrastructure for synchronizing arbitrary many models. In our specific case, these source models emerged from ecore models and were converted to SCROLL code automatically through a code generator developed at our chair. The main problem then boiled down to implementing a transformation compartment to perform the transformation. As mentioned earlier even this process might be carried out (semi-)automatically and constitutes a main line for further research. In addition, the optimization of SCROLL is another task for the future so that it performs role bindings in an optimal way.

Acknowledgements

This work has been funded by the German Research Foundation within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907), the research project "Rule-Based Invasive Software Composition with Strategic Port-Graph Rewriting" (RISCOS) and by the German Federal Ministry of Education and Research within the project "OpenLicht'.

References

[BD77] Charles W. Bachman and Manilal Daya. The Role Concept in Data Models. In *3rd International Conference on Very Large Data Bases*, volume 3, pages 464–476. VLDB Endowment, 1977.

[GDH19] Antonio Garcia-Dominguez and Georg Hinkel. Truth Tables to Binary Decision Diagrams. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 12th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2019) federation of conferences*, CEUR Workshop Proceedings. CEUR-WS.org, July 2019.

[KBRA16] Thomas Kühn, Kay Bierzynski, Sebastian Richly, and Uwe Aßmann. FRaMED: Full-fledge Role Modeling Editor (Tool Demo). In *SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 132–136, New York, NY, USA, 2016. ACM.

[KLG+14] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A Metamodel Family for Role-Based Modeling and Programming Languages. In *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 141–160. Springer, 2014.

[LA15] Max Leuthäuser and Uwe Aßmann. Enabling View-based Programming with SCROLL: Using Roles and Dynamic Dispatch for Etablishing View-based Programming. In *Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, MORSE/VAO '15, pages 25–33, New York, NY, USA, 2015. ACM.

[Ste00] Friedrich Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering*, 35(1):83–106, 2000.

[WSK+18] Christopher Werner, Hendrik Schön, Thomas Kühn, Sebastian Götz, and Uwe Aßmann. Role-Based Runtime Model Synchronization. In *44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 306–313, Aug 2018.

Truth Tables to Binary Decision Diagrams in Modern ATL

Dennis Wagelaar
Corilus
Vilvoorde, Belgium
dennis.wagelaar@corilus.be

Théo Le Calvar
LERIA, Université d'Angers
Angers, France
theo.lecalvar@univ-angers.fr

Frédéric Jouault
ERIS, ESEO-TECH
Angers, France
frederic.jouault@eseo.fr

Abstract

Model transformation technology has evolved over the last 15 years, notably regarding scalability and performance. The Truth Tables to Binary Decision Diagrams transformation was written for an early version of ATL roughly 13 years ago. At that time, performance was not as much of a concern as it is today. Not only were execution engines slower than now, but they also did not provide as many optimization opportunities. Consequently, in its original form, this transformation does not scale well to large models. It remains slow, even when using EMFTVM, the state-of-the-art ATL virtual machine. In this work, we show that by leveraging the profiler, and carefully optimizing the transformation code, significantly improved performance can be achieved. Our updated solution scales up to the largest generated model ($\sim$ 40 MB), which is transformed in about 23 seconds on a modern desktop (Ryzen 5 1600X with 16 GB RAM running Fedora 29): several hundred times faster than the original code.

1 Introduction

This paper presents the ATL/EMFTVM solution to the offline case of the Transformation Tool Contest[1] 2019: Truth Tables to Binary Decision Diagrams (TT2BDD)[2] [GDH19]. This case was developed from an ATL transformation published in the ATL Transformation Zoo [Sav06] roughly 13 years ago in 2006: TT2BDD, a model transformation initially written for the first ATL virtual machine ever released. Since that time, two generations of ATL virtual machines have been developed: EMFVM around 2007, and EMFTVM[3] [WTCJ11] from 2011 on.

The resources for the case include the original ATL code from 2006, but run it with EMFTVM. This results in better performance than using the virtual machine of the time, but it does not solve the issues that are in the transformation code. Our updated solution[4] is called ATLEMFTVMImproved, and mostly consists of optimizations to the transformation code. These optimizations are described in the following section.

In: Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava (eds.): Proceedings of the 12th Transformation Tool Contest, Eindhoven, The Netherlands, 19-07-2019, published at http://ceur-ws.org

[1] `https://www.transformation-tool-contest.eu/`

[2] GitHub repository with accompanying resources: `https://github.com/TransformationToolContest/ttc2019-tt2bdd`

[3] Documented at `https://wiki.eclipse.org/ATL/EMFTVM`.

[4] GitHub repository with our solution: `https://github.com/dwagelaar/ttc2019-tt2bdd`

2 Optimizations

In order to optimize the TT2BDD transformation, we used the EMFTVM built-in profiler. This lead us to the three following optimizations:

- **Leveraging helper attributes caching.** ATL provides two kinds of helpers: helper operations, and helper attributes. Helper attributes are basically similar to parameterless helper operations with a significant performance-related difference: their result is cached. Therefore, multiple accesses do not result in multiple computations. The `getTree` helper operation was thus changed into a `tree` helper attribute. With a lower performance impact, the `getNode` helper operation was also changed into a `node` helper attribute.
- **Applying the object indexing pattern[5].** This pattern uses a Map in order to avoid expensive lookups that can be precomputed. A typical example is the navigation of missing opposite references.
- **Leveraging Maps.** In the `getPartition` helper operations, the original row-accessing code is quadratic: a use of the `exists` iterator in the body of a `select` iterator. We instead compute two Maps: one for each of the possible `true` and `false` values. These Maps are indexed by ports. Moreover, these Maps are computed in helper attributes, and are therefore cached and computed only once per context. Because EMFTVM uses a HashMap to implement Maps, the resulting code has a linear time complexity.

Another optimization was performed in the transformation launcher: module loading code was moved into the initialization phase.

The optimizations were applied after repeated measurements with the ATL/EMFTVM profiler. This profiler can be enabled by checking the "Display Profiling Data" box in the Eclipse Run Configuration dialog. Listing 6 in Appendix shows the output of running the ATL reference solution (before any optimization) against the "GeneratedI8O2Seed68.ttmodel". Typically, the maximum gains can be achieved by addressing the top lines of the profiler output, in this case the `getPartition` and `getTree` helper operations. The top line of the profiler output amounts to 21.45% of the total measured runtime of the transformation. It represents a qualified portion of the `getPartition` helper operation: the "@0@0" qualifier designates the first closure within the first closure of the `getPartition` helper body, which is invoked 41811968 times.

Listing 1 shows the `getPartition` helper operation. The first closure within the first closure is the body of the `exists` iterator within the `select` iterator (line 9). In this case, the fact that so much runtime is spent here is due to the high amount of invocations.

```
1  helper def:
2     getPartition(rows : Sequence(TT!Row), port : TT!Port)
3        : TupleType( zeroPart : Sequence(TT!Row) , onePart : Sequence(TT!Row) ) =
4
5        -- Select the rows for which the port is false
6        let _zeroPart : Sequence(TT!Row) =
7           rows->select(r |
8              r.cells->exists(c |
9                 c.port = port and c.value = false
10             )
11          ) in
12
13       -- Select the rows for which the port is true
14       let _onePart : Sequence(TT!Row) =
15          rows->select(r |
16             r.cells->exists(c |
17                c.port = port and c.value = true
18             )
19          ) in
20
21       -- Build the resulting tuple
22       Tuple{
23          zeroPart = _zeroPart,
24          onePart = _onePart
25       };
```

Listing 1: The `getPartition` helper operation

Instead of trying to optimize directly at this level, we first trace back where the `getPartition` helper is invoked: it is invoked only from within the `getTree` helper operation. The `getTree` helper operation is also

[5] `https://wiki.eclipse.org/ATL/Design_Patterns#Object_indexing`

responsible for a large portion of the transformation runtime: the main helper body is responsible for 8.50% of the measured runtime, and is invoked 718080 times (line 7 of Listing 6). There are two `getTree` helper operations: one with input parameters, and one without. The one with input parameters invokes itself recursively, and the one without input parameters only invokes the one with parameters.

Listing 2 shows the `getTree` helper without input parameters. This helper operation is invoked on the `TT!TruthTable` context, of which there is always one instance in an input model. Yet, in 6, line 16, it shows up with 2816 invocations. If we convert this helper operation into a helper attribute, its body will be executed only once for each `TT!TruthTable` instance, and any subsequent invocations will be retrieved from cache.

```
1 helper context TT!TruthTable def:
2    getTree()
3       : TupleType( cell : TT!Cell , zeroSubtree : OclAny , oneSubtree : OclAny ) =
4       thisModule.getTree(self.rows, self.ports->select(p | p.oclIsKindOf(TT!InputPort)));
```

Listing 2: The `getTree` helper operation

Commit c1208ae[6] contains the first performance optimization: convert the parameterless `getTree` helper operation into the `tree` helper attribute. Listing 7 shows the profiler output after applying this optimization. There is only one invocation of the `tree` helper attribute body on line 21. As a result, the amount of runtime spent in the remaining `getTree` helper operation and the `getPartition` helper operation has also been drastically reduced. The total runtime has been reduced from 69.832018 seconds to 0.703755 seconds.

Now, the top line in the profiler output points to the `findCell` helper operation. `findCell` is invoked by the `getNode` helper operation. `getNode` is also a parameterless helper operation, which can easily be converted to a helper attribute to reduce the amount of times its body is invoked. Commit dd9b976[7] does just that, and Listing 8 shows the profiler output after applying this optimization. The 2815 invocations of `getNode` on line 10 of Listing 7 have become 2560 invocations of the `node` helper attribute on line 10 of Listing 8. Not a significant improvement this time, but it has been achieved with very little effort.

That leaves the `findCell` helper at the top of the list with 620415 invocations, responsible for 87.77% of the measured runtime. Listing 3 shows this `findCell` helper. Its purpose is to find the corresponding tree node for a given TT cell. Ideally, it is invoked once per cell, but the representation of the in-memory tree structure makes that we cannot simply convert this operation into an attribute. We therefore resort to the object indexing pattern, which computes a Map of cells to their tree nodes.

```
1  helper def:
2     findCell(cell : TT!Cell, tree : TupleType( cell : TT!Cell, zeroSubtree : OclAny, oneSubtree : OclAny ))
3        : TupleType( cell : TT!Cell , zeroSubtree : OclAny , oneSubtree : OclAny ) =
4
5        if tree.cell = cell then
6           tree
7        else if tree.zeroSubtree.oclIsKindOf(TT!Row) then
8           if tree.oneSubtree.oclIsKindOf(TT!Row) then
9              -- Both subtrees are leaf nodes
10             OclUndefined
11          else
12             -- Only the subtree 1 is not a leaf
13             thisModule.findCell(cell,tree.oneSubtree)
14          endif
15       else
16          let tryInZero : OclAny = thisModule.findCell(cell,tree.zeroSubtree) in
17          if tree.oneSubtree.oclIsKindOf(TT!Row) then
18             -- Only the subtree 0 is not a leaf
19             tryInZero
20          else if tryInZero.oclIsUndefined() then
21             -- Both subtrees are non-leaves, but subtree 0 did not produce any results
22             thisModule.findCell(cell,tree.oneSubtree)
23          else
24             -- Both subtrees are non-leaves, and subtree 0 has produced results
25             tryInZero
26          endif endif
27       endif endif;
```

Listing 3: The `findCell` helper operation

[6] https://github.com/dwagelaar/ttc2019-tt2bdd/commit/c1208aebc2e3a89601411b66c83446f555dd9fe6
[7] https://github.com/dwagelaar/ttc2019-tt2bdd/commit/dd9b976fb79d95bb11748bb96d5e8326fa43feae

Listing 4 shows the new `nodesByCell` helper attribute, with its companion `collectAllNodes` helper operation. `nodesByCell` uses the `mappedBySingle` built-in helper operation introduced by EMFTVM to convert a list of tree nodes into a map of cells to nodes. Whereas it is trivial to find the accompanying cell for a given tree node, there is no direct way to find the tree node for a given cell. `mappedBySingle` enables one to quickly reverse navigate a given EMF `EReference`. The `collectAllNodes` helper operation serves to flatten the tree of nodes into a `Sequence` of nodes, such that it can be consumed by `mappedBySingle`.

```
1  helper context TT!TruthTable def:
2          nodesByCell
3                  : Map(TT!Cell, TupleType( cell : TT!Cell, zeroSubtree : OclAny, oneSubtree : OclAny )) =
4
5                  thisModule.collectAllNodes(self.tree)
6                          ->mappedBySingle(node | node.cell);
7
8  helper def:
9          collectAllNodes(tree : TupleType( cell : TT!Cell, zeroSubtree : OclAny, oneSubtree : OclAny ))
10                 : Sequence(TupleType( cell : TT!Cell , zeroSubtree : OclAny , oneSubtree : OclAny )) =
11
12                 if tree.zeroSubtree.oclIsKindOf(TT!Row) then
13                         if tree.oneSubtree.oclIsKindOf(TT!Row) then
14                 -- Both subtrees are leaf nodes
15                 Sequence{}
16                 else
17                 -- Only the subtree 1 is not a leaf
18                 thisModule.collectAllNodes(tree.oneSubtree)
19                         endif
20                 else
21                         if tree.oneSubtree.oclIsKindOf(TT!Row) then
22                 -- Only the subtree 0 is not a leaf
23                 thisModule.collectAllNodes(tree.zeroSubtree)
24                         else
25                 -- Both subtrees are non-leaves
26                                 thisModule.collectAllNodes(tree.zeroSubtree)->union(
27                                         thisModule.collectAllNodes(tree.oneSubtree))
28                         endif
29                 endif
30                 ->prepend(tree);
```

Listing 4: The `findCell` helper operation

Listing 9 shows the profiler output after applying the object indexing pattern: total measured runtime has been reduced from 0.670487 seconds to 0.086927 seconds, which is significant. The remaining top entries in the profiling output are `getPartition` and `getTree` (and contained closures thereof). We will focus on the top entry (line 3), which is the first closure within the first closure of `getPartition`. Listing 1 shows the `getPartition` helper operation, our first example, of which line 9 represents the first closure within the first closure. Along with line 5 of the profiler output – the first closure within the second closure (Listing 1 line 17) – this closure stands out through its high amount of invocations (14848), and resulting percentage of the total measured runtime (12.37%).

In commit 6fcd025[8], we again apply the object indexing pattern to quickly retrieve cells by their port, also prefiltered by their value (true or false). Listing 5 shows the improved version of the `getPartition` helper. The double nesting of closures has been eliminated, and instead two Maps are created for each row, containing the `true` cells mapped by their port and the `false` cells by their port.

Listing 10 shows the profiler output after this optimization. The remaining entries in this output show little opportunity for further optimization: either their percentage in the total measured runtime is very low, or the number of invocations is very low (i.e. not higher than the amount of model elements). As such, we have decided not to optimize further at this point. Overall, we have achieved a speedup of 912 times.

3 Conclusion

The state of the art EMFTVM was able to run an old ATL transformation. Although, it cannot automatically optimize it, its built-in profiler makes it possible to quickly spot performance bottlenecks and fix them by making relatively simple changes such as turning parameterless helper operations into helper attributes, or applying well-documented patterns such as the object indexing pattern. This paper has also illustrated how to interpret the profiler output, and how to use it in order to apply performance optimizations.

[8] `https://github.com/dwagelaar/ttc2019-tt2bdd/commit/6fcd025952451461a2affb415a75c54f90f3562b`

```
helper def:
   getPartition(rows : Sequence(TT!Row), port : TT!Port)
      : TupleType( zeroPart : Sequence(TT!Row) , onePart : Sequence(TT!Row) ) =

      -- Select the rows for which the port is false
      let _zeroPart : Sequence(TT!Row) =
         rows->reject(r |
            r.falseCellsByPort.get(port).oclIsUndefined()
         ) in

      -- Select the rows for which the port is true
      let _onePart : Sequence(TT!Row) =
         rows->reject(r |
            r.trueCellsByPort.get(port).oclIsUndefined()
         ) in

      -- Build the resulting tuple
      Tuple{
         zeroPart = _zeroPart,
         onePart = _onePart
      };

helper context TT!Row def: trueCellsByPort : Map(TT!Port, Set(TT!Cell)) =
        self.cells
                ->select(c | c.value)
                ->mappedBy(c | c.port);

helper context TT!Row def: falseCellsByPort : Map(TT!Port, Set(TT!Cell)) =
        self.cells
                ->reject(c | c.value)
                ->mappedBy(c | c.port);
```

Listing 5: The improved `getPartition` helper operation

References

[GDH19] Antonio Garcia-Dominguez and Georg Hinkel. Truth Tables to Binary Decision Diagrams. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 12th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2019) federation of conferences*, CEUR Workshop Proceedings. CEUR-WS.org, July 2019.

[Sav06] Guillaume Savaton. Truth Tables to Binary Decision Diagrams. ATL Transformations, `https://www.eclipse.org/atl/atlTransformations/#TT2BDD`, February 2006. Last accessed on 2019-05-14. Archived on `http://archive.is/HdoHM`.

[WTCJ11] Dennis Wagelaar, Massimo Tisi, Jordi Cabot, and Frédéric Jouault. Towards a General Composition Semantics for Rule-based Model Transformation. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 623–637, Berlin, Heidelberg, 2011. Springer-Verlag.

Appendix

```
Duration (sec.) Duration (%)    Invocations     Operation
14,974607    21,45  41811968    static EMFTVM!ExecEnv::getPartition(...) : Tuple@0@0
14,942565    21,40  41811968    static EMFTVM!ExecEnv::getPartition(...) : Tuple@1@0
 6,402944     9,17   5767168    static EMFTVM!ExecEnv::getPartition(...) : Tuple@0
 6,379739     9,14   5767168    static EMFTVM!ExecEnv::getPartition(...) : Tuple@1
 5,932174     8,50    718080    static EMFTVM!ExecEnv::getTree(...) : Tuple
 4,628028     6,63   5767168    static EMFTVM!ExecEnv::getTree(...) : Tuple@0@0
 4,183250     5,99  25952256    static EMFTVM!ExecEnv::getTree(...) : Tuple@0@0@0
 1,597109     2,29   5049088    static EMFTVM!ExecEnv::getTree(...) : Tuple@1
 1,380862     1,98   5049088    TT!TruthTable::getTree() : Tuple@0
 0,732114     1,05    653055    static EMFTVM!ExecEnv::findCell(...) : Tuple
 0,707898     1,01    718080    static EMFTVM!ExecEnv::getTree(...) : Tuple@0
 0,701314     1,00    718080    static EMFTVM!ExecEnv::getPartition(...) : Tuple
 0,010936     0,02      2815    TT!Cell::getNode() : Tuple
 0,008518     0,01      2816    TT!TruthTable::getTree() : Tuple
 0,005499     0,01       255    rule Cell2Subtree@applier
 0,004568     0,01      2560    rule Cell2Subtree@matcher
 0,003421     0,00      2560    rule Cell2Assignment@matcher
 0,002911     0,00       256    rule Row2Leaf@applier
 0,002905     0,00      2560    rule Row2Leaf@applier@0
 0,001704     0,00       512    rule Cell2Assignment@applier
 0,000131     0,00         1    rule TruthTable2BDD@applier
 0,000015     0,00         8    rule InputPort2InputPort@applier
 0,000004     0,00         2    rule OutputPort2OutputPort@applier
 0,000001     0,00         1    static EMFTVM!ExecEnv::init() : Object
 0,000000     0,00         1    static EMFTVM!ExecEnv::main() : Object
Timing data:
        Loading finished at 0,008558 seconds (duration: 0,008558 seconds)
        Matching finished at 63,503997 seconds (duration: 63,495439 seconds)
        Applying finished at 69,830402 seconds (duration: 6,326405 seconds)
        Post-applying finished at 69,830517 seconds (duration: 0,000115 seconds)
        Recursive stage finished at 69,830529 seconds (duration: 0,000012 seconds)
        Execution finished at 69,832018 seconds (duration: 0,001501 seconds)
```

Listing 6: ATL/EMFTVM Profiler output 1

```
Duration (sec.) Duration (%)    Invocations     Operation
0,613147   88,17 653055 static EMFTVM!ExecEnv::findCell(cell: TT!Cell, tree: Tuple) : Tuple
0,010024    1,44  14848 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple@0@0
0,009428    1,36    255 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple
0,009250    1,33  14848 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple@1@0
0,004373    0,63   2048 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple@1
0,004244    0,61   2048 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple@0
0,003874    0,56    255 rule Cell2Subtree@applier
0,003748    0,54   2815 TT!Cell::getNode() : Tuple
0,003501    0,50   2048 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@0@0
0,003118    0,45   2560 rule Cell2Assignment@matcher
0,002839    0,41    256 rule Row2Leaf@applier
0,002664    0,38   2560 rule Row2Leaf@applier@0
0,002371    0,34   2560 rule Cell2Subtree@matcher
0,002108    0,30   9216 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@0@0@0
0,001689    0,24    255 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple
0,001606    0,23   1793 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@1
0,001535    0,22    512 rule Cell2Assignment@applier
0,001383    0,20    255 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@0
0,001256    0,18   1793 TT!TruthTable::tree: Tuple@0
0,000070    0,01      1 rule TruthTable2BDD@applier
0,000028    0,00      1 TT!TruthTable::tree: Tuple
0,000015    0,00      8 rule InputPort2InputPort@applier
0,000004    0,00      2 rule OutputPort2OutputPort@applier
0,000000    0,00      1 static EMFTVM!ExecEnv::init() : Object
0,000000    0,00      1 static EMFTVM!ExecEnv::main() : Object
Timing data:
        Loading finished at 0,007563 seconds (duration: 0,007563 seconds)
        Matching finished at 0,660266 seconds (duration: 0,652703 seconds)
        Applying finished at 0,702967 seconds (duration: 0,042700 seconds)
        Post-applying finished at 0,703010 seconds (duration: 0,000043 seconds)
        Recursive stage finished at 0,703021 seconds (duration: 0,000011 seconds)
        Execution finished at 0,703755 seconds (duration: 0,000745 seconds)
```

Listing 7: ATL/EMFTVM Profiler output 2

```
Duration (sec.) Duration (%)    Invocations     Operation
0,580876   87,77 620415 static EMFTVM!ExecEnv::findCell(cell: TT!Cell, tree: Tuple) : Tuple
0,009810    1,48  14848 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple@0@0
0,009739    1,47  14848 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple@1@0
0,009731    1,47    255 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple
0,004357    0,66   2048 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple@0
0,004235    0,64   2048 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple@1
0,003370    0,51   2048 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@0@0
0,003306    0,50   2560 TT!Cell::node: Tuple
0,003206    0,48   2560 rule Cell2Subtree@matcher
0,003112    0,47    255 rule Cell2Subtree@applier
0,002909    0,44   2560 rule Row2Leaf@applier@0
0,002857    0,43    256 rule Row2Leaf@applier
0,002754    0,42   2560 rule Cell2Assignment@matcher
0,002142    0,32   9216 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@0@0@0
0,001729    0,26    255 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple
0,001663    0,25    512 rule Cell2Assignment@applier
0,001626    0,25   1793 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@1
0,001321    0,20   1793 TT!TruthTable::tree: Tuple@0
0,001057    0,16    255 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@0
0,000074    0,01      1 rule TruthTable2BDD@applier
0,000024    0,00      1 TT!TruthTable::tree: Tuple
0,000017    0,00      2 rule OutputPort2OutputPort@applier
0,000015    0,00      8 rule InputPort2InputPort@applier
0,000000    0,00      1 static EMFTVM!ExecEnv::init() : Object
0,000000    0,00      1 static EMFTVM!ExecEnv::main() : Object
Timing data:
        Loading finished at 0,007919 seconds (duration: 0,007919 seconds)
        Matching finished at 0,657334 seconds (duration: 0,649415 seconds)
        Applying finished at 0,669723 seconds (duration: 0,012390 seconds)
        Post-applying finished at 0,669771 seconds (duration: 0,000048 seconds)
        Recursive stage finished at 0,669782 seconds (duration: 0,000011 seconds)
        Execution finished at 0,670487 seconds (duration: 0,000716 seconds)
```

Listing 8: ATL/EMFTVM Profiler output 3

```
Duration (sec.) Duration (%)    Invocations     Operation
0,009697   12,37  14848 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple@0@0
0,009125   11,64    255 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple
0,008714   11,12  14848 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple@1@0
0,004266    5,44   2048 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple@0
0,004040    5,15   2048 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple@1
0,003325    4,24   2048 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@0@0
0,003175    4,05    255 rule Cell2Subtree@applier
0,003014    3,84   2560 rule Cell2Assignment@matcher
0,002862    3,65    256 rule Row2Leaf@applier
0,002754    3,51   2560 rule Row2Leaf@applier@0
0,002372    3,03   2560 TT!Cell::node: Tuple
0,002005    2,56   9216 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@0@0@0
0,001887    2,41    255 static EMFTVM!ExecEnv::collectAllNodes(tree: Tuple) : Sequence
0,001874    2,39   2560 rule Cell2Subtree@matcher
0,001747    2,23    512 rule Cell2Assignment@applier
0,001722    2,20    255 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple
0,001419    1,81   1793 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@1
0,001202    1,53   1793 TT!TruthTable::tree: Tuple@0
0,000994    1,27    255 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@0
0,000444    0,57      1 TT!TruthTable::nodesByCell: Map
0,000103    0,13    255 TT!TruthTable::nodesByCell: Map@0
0,000056    0,07      1 rule TruthTable2BDD@applier
0,000024    0,03      1 TT!TruthTable::tree: Tuple
0,000014    0,02      8 rule InputPort2InputPort@applier
0,000003    0,00      2 rule OutputPort2OutputPort@applier
0,000000    0,00      1 static EMFTVM!ExecEnv::init() : Object
0,000000    0,00      1 static EMFTVM!ExecEnv::main() : Object
Timing data:
        Loading finished at 0,007728 seconds (duration: 0,007728 seconds)
        Matching finished at 0,072620 seconds (duration: 0,064892 seconds)
        Applying finished at 0,086084 seconds (duration: 0,013464 seconds)
        Post-applying finished at 0,086153 seconds (duration: 0,000068 seconds)
        Recursive stage finished at 0,086164 seconds (duration: 0,000011 seconds)
        Execution finished at 0,086927 seconds (duration: 0,000775 seconds)
```

Listing 9: ATL/EMFTVM Profiler output 4

```
Duration (sec.) Duration (%)    Invocations     Operation
0,011395   17,15    255 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple
0,003796    5,71   2048 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@0@0
0,003340    5,03    255 rule Cell2Subtree@applier
0,003258    4,90   2560 rule Cell2Assignment@matcher
0,002754    4,15    256 TT!Row::trueCellsByPort: Map
0,002460    3,70   2048 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple@1
0,002439    3,67    256 rule Row2Leaf@applier
0,002417    3,64   2560 rule Row2Leaf@applier@0
0,002390    3,60   9216 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@0@0@0
0,002278    3,43   2560 TT!Cell::node: Tuple
0,002226    3,35   2048 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple@0
0,002001    3,01    255 static EMFTVM!ExecEnv::collectAllNodes(tree: Tuple) : Sequence
0,001973    2,97    256 TT!Row::falseCellsByPort: Map
0,001840    2,77    255 static EMFTVM!ExecEnv::getPartition(rows: Sequence, port: TT!Port) : Tuple
0,001804    2,71   2560 rule Cell2Subtree@matcher
0,001580    2,38    512 rule Cell2Assignment@applier
0,001364    2,05   1793 TT!TruthTable::tree: Tuple@0
0,001340    2,02   1793 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@1
0,001212    1,82   2560 TT!Row::trueCellsByPort: Map@0
0,001167    1,76    255 static EMFTVM!ExecEnv::getTree(rows: Sequence, usablePorts: Sequence) : Tuple@0
0,001120    1,69   2560 TT!Row::falseCellsByPort: Map@0
0,000572    0,86   1285 TT!Row::trueCellsByPort: Map@1
0,000502    0,76   1275 TT!Row::falseCellsByPort: Map@1
0,000422    0,63      1 TT!TruthTable::nodesByCell: Map
0,000098    0,15    255 TT!TruthTable::nodesByCell: Map@0
0,000056    0,08      1 rule TruthTable2BDD@applier
0,000028    0,04      1 TT!TruthTable::tree: Tuple
0,000012    0,02      8 rule InputPort2InputPort@applier
0,000004    0,01      2 rule OutputPort2OutputPort@applier
0,000000    0,00      1 static EMFTVM!ExecEnv::init() : Object
0,000000    0,00      1 static EMFTVM!ExecEnv::main() : Object
Timing data:
        Loading finished at 0,008821 seconds (duration: 0,008821 seconds)
        Matching finished at 0,063594 seconds (duration: 0,054773 seconds)
        Applying finished at 0,075310 seconds (duration: 0,011716 seconds)
        Post-applying finished at 0,075353 seconds (duration: 0,000043 seconds)
        Recursive stage finished at 0,075369 seconds (duration: 0,000016 seconds)
        Execution finished at 0,076570 seconds (duration: 0,001217 seconds)
```

Listing 10: ATL/EMFTVM Profiler output 5

Part II.

BibtexXML to Docbook Consistency Case Live Case

The TTC 2019 Live Case: BibTeX to DocBook

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

Georg Hinkel
Tecan Software Competence Center
55252, Mainz-Kastel, Germany
georg.hinkel@tecan.com

Abstract

The initial transformation of a model into another model is only the first step. After the creation of the target model, it may be manually changed and the consistency with the source model may be lost or obscured. Ideally, transformation tools should have a way to check the degree of consistency between the source model and the current version of the destination model. This case presents such a scenario for a small transformation, with an automated mutation tool which will introduce changes that may or may not impact consistency. The aim of this case is to evaluate the speed and verbosity of the inter-model consistency checking in the state of the art.

1 Introduction

This live case is based on the original ATL Zoo [2] BibTeX to DocBook transformation, where a simplified version of the BibTeX reference manager's data model (shown in Figure 1) is transformed to a simplification of the DocBook document typesetting tool (shown in Figure 2).

The transformation consists of these mappings:

- From the root BibTeXFile, a DocBook is created. The DocBook has a Book with an Article titled "BibTeXML to DocBook", which has in turn four Sect1 instances: "References List", "Author List", "Titles List" and "Journals List".
- Any BibTeX Author is listed in the "Authors List" as a Para.
- Any BibTeXEntry is listed in the "References List", including its unique identifier and any available information.
- For each TitledEntry, a Para with its title is added to the "Titles List"
- For each BibTeX Article, a Para with its journal name is added to the "Journals List".
- The "Author List", "Titles List", and "Journals List" are all sorted lexicographically in ascending order.

The transformation is somewhat different in that it involves some sorting. The original ATL-based implementation worked well for instances with 10, 100 and 1000 entries, but seems to struggle with instances with 10000 entries.

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

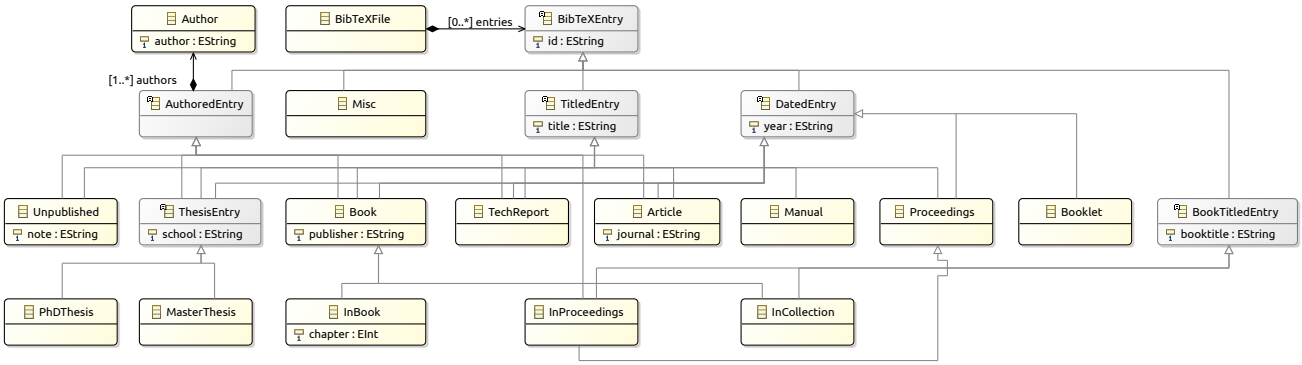


Figure 1: Class diagram for the input BibTeXML metamodel

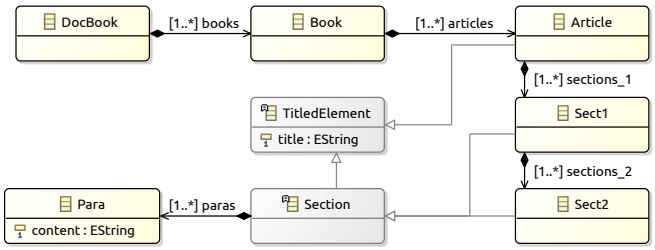


Figure 2: Class diagram for the target DocBook metamodel

Still, a more interesting problem is the case when the target model continues to be worked on after the initial transformation. The DocBook document would be given to an editor, which would reorganise sections, add paragraphs in the middle with further commentary and perhaps extend some of the text itself. The task is to ensure in an efficient manner that these manual editions have not impacted the consistency of the DocBook model with the original BibTeX model.

Ideally, the transformation tool or the editing environment would give some infrastructure to tackle this. However, for many tools, the approach seems to be only feasible through the creation of an external consistency checker. This case is to evaluate the current state of the art in out-of-the-box after-the-fact consistency checking. To do so, the case provides a generator which can produce source models of arbitrary size, a repackaged version of the original ATL transformation, and a mutator which can make a number of changes to the model, some of which may impact consistency. These resources were made available on Github[1] before the start of STAF 2019.

The rest of the document is structured as follows: Section 2 describes the structure of the live case. Section 3 describes the proposed tasks for this live case. 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 Case Structure

The case is intended to review the different approaches for checking after-the-fact inter-model consistency between a BibTeX model and a DocBook model. The process is roughly as follows:

1. The BibTeX model is generated randomly to a certain size, by the included *generator* in the `models/generator.jar` JAR. The generator uses a Java port of the Ruby Faker[2] library to produce pseudorandom data given a seed. A number of random models (sizes 10, 100, 1000 and 10000) were generated in advance and included in the case resources.

2. The BibTeX model is transformed automatically to DocBook by the repackaged version of the original ATL transformation in the `bibtex2docbook.jar` JAR. This transformation deals well with models up to 1000 entries, but struggles with larger models due to recomputation of intermediate results.

[1] https://github.com/TransformationToolContest/ttc2019-live
[2] https://github.com/DiUS/java-faker

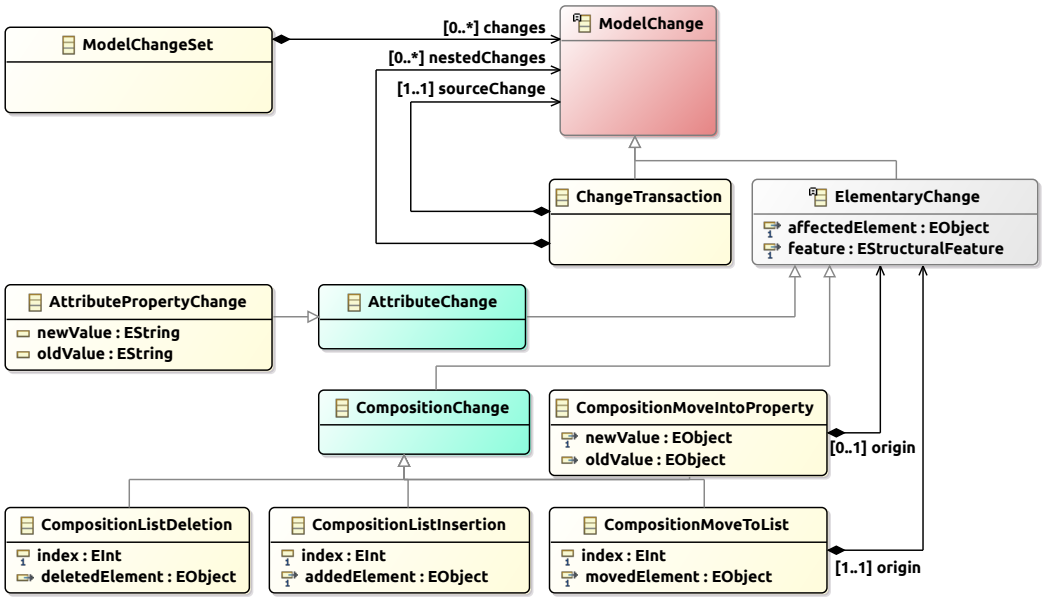


Figure 3: Class diagram for the used subset of the Changes metamodel

3. The DocBook model is edited, in this case with the automated *mutator* in the `models/mutator.jar` JAR. The mutator will operate on a DocBook file, creating a set of folders whose path will start with the specified prefix, adding `-N` from 1 to `nMutants`. Each folder will contain the mutated DocBook model, as well as a *change model* explaining what was done to the model: Figure 3 shows that such models have a ModelChangeSet as the root, with a number of ModelChange instances of various types. The DocBook model will have gone through a number of random mutations according to a seed: if unspecified, the seed will be based on the current system time.

   The mutator has a number of predefined *mutation operators* that will modify the model:

   - Swapping paragraphs: this should break consistency in terms of sorting, but it will not result in missing information.
   - Swapping sections should not break consistency.
   - Deleting paragraphs/sections should break consistency.
   - Appending text to a paragraph should not break consistency.
   - Adding a new paragraph: unless it happens to match one of the authors, titles, or journals in its Sect1, it should not break consistency.

   The mutated models were created in advance before the contest. There are three sets of mutated models from the generated 10/100/1000-entry models: one with a single mutation, one with two mutations, and one with three mutations.

4. A *consistency checker* would take any combination of the previous artifacts (source BibTeX, fresh DocBook, mutated DocBook, change model) and make a judgment about whether the mutated DocBook is still consistent or not.

   If issues are found, it should point to the element in the source model which lacks a proper mapping on the other side, or the element in the target model which is not mapped correctly from the source model (e.g. it is not sorted anymore).

   The case resources include a set of expected results from the reference EVL consistency checker. However, the concrete definition of the consistency requirements has proven to be trickier to formalize than expected. This was raised by several case authors, and in fact one of the solutions considered creating a DSL for expressing inter-model consistency.

3 Task Description

The case had an optional and a mandatory task:

- The optional task was to re-implement or improve the original transformation itself, in a way that lent itself better to after-the-fact consistency checking. A transformation tool may have better support for this, or ATL could be made to deal better with larger versions of this model.
- The mandatory task was to check for the consistency of the source BibTeX against the mutated DocBook models in the `models` directory, and report this information as efficiently and clearly as possible to the user. Ideally, this should be possible without a full re-run of the transformation. To be considered for the performance-related awards, solutions had to use the benchmarking framework in Section 4.

A reference solution based on the Epsilon Validation Language was provided. This implementation did not re-run any transformations: instead, it did a two-way consistency validation by checking from the BibTeX BibTeXFile, Author, BibTeXEntry, TitledEntry, and Article types, and from the DocBook Para types. The implementation required 98 lines of EVL code and 113 lines of Java framework integration code, and does not use the change models at all.

Solutions could focus on efficiency, conciseness, or clarity of presentation to the user. Solutions that can operate straight from the definition of the transformation (i.e. without a separate consistency checker) would be preferred. The call for solutions also invited solution authors to consider other desirable attributes, e.g. verifiability.

4 Benchmark Framework

If focusing on performance, the solution authors had to integrate their solution with the provided benchmark framework. It is based on the framework in the TTC 2017 Smart Grid case [1], and supports the automated build and execution of solutions. The benchmark consisted of three phases:

1. **Initialization**, which involved setting up the basic infrastructure (e.g. loading metamodels). These measurements are optional.
2. **Load**, which loaded the input models.
3. **Run**, which found the consistency violations in the mutated DocBook model.

4.1 Solution requirements

Each solution had to print to the standard output a line with the following fields, separated by semicolons (";"):

- **Tool**: name of the tool.
- **MutantSet**: set of mutants used ("single", "double" or "triple").
- **Source**: base name of the input BibTeX model (e.g. "random10.bibtex").
- **Mutant**: integer starting at 1, identifying the mutant model within this set.
- **RunIndex**: index of the run of this combination of tools and inputs.
- **PhaseName**: name of the phase being run.
- **MetricName**: the name of the metric. It may be the used **Memory** in bytes, the wall clock **Time** spent in integer nanoseconds, or the number of consistency **Problems** found in the mutated DocBook model.

To enable automatic execution by the benchmark framework, solutions were in the form of a subfolder within the `solutions` folder of the main repository[3], with a `solution.ini` file stating how the solution should be built and how it should be run. As an example, the `solution.ini` file for the reference solution is shown on Listing 1. In the `build` section, the `default` option specifies the command to build and test the solution, and

[3] https://github.com/TransformationToolContest/ttc2019-live

Listing 1: `solution.ini` file for the reference Epsilon solution

```
1 [build]
2 default=true
3 skipTests=true
4
5 [run]
6 cmd=JAVA_OPTS="-Xms4g" java -jar epsilon.jar
```

the `skipTests` option specifies the command to build the solution while skipping unit tests. In the `run` section, the `cmd` option specifies the command to run the solution.

The repetition of executions as defined in the benchmark configuration was 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 could not save intermediate data between different runs: each run had to be entirely independent.

The name and absolute path of the input model, the run index and the name of the tool were passed using environment variables `Tool`, `MutantSet`, `SourcePath`, `Mutant`, `MutantPath`, and `RunIndex`.

4.2 Running the benchmark

The benchmark framework only required Python 3.3 to be installed. Solutions could use any languages or frameworks, as long as they could run without human input. Since all the performance-oriented solutions this year were compatible with GNU/Linux, it was possible for the case authors to create a `Dockerfile` with all solutions built in, for the sake of reproducibility. The resulting image is available on Docker Hub[4], and it is automatically rebuilt on any push to the repository.

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

The evaluation operated on several dimensions:

- How efficient was the approach in time and space (memory)? The reference ATL solution struggled with large models, and the reference solution was not been designed with performance in mind.
- Was consistency checking directly supported by the transformation approach? Many tools lack this capability, though it might be interesting as an additional execution mode if the target model has seen manual changes since it was generated.
- How informative and accessible was the feedback that can be provided by the approach?

Authors were invited to make submissions targeting other quality attributes.

References

[1] Georg Hinkel. The TTC 2017 Outage System Case for Incremental Model Views. In *Proceedings of the 10th Transformation Tool Contest*, volume 2026, pages 3–12, Marburg, Germany, July 2017. CEUR-WS.org.

[2] Guillaume Savaton. BibTeXML to DocBook, ATL Transformations. https://www.eclipse.org/atl/atlTransformations/#BibTeXML2DocBook, February 2006. Last accessed on 2019-07-15. Archived on http://archive.is/HdoHM.

[4] https://hub.docker.com/r/bluezio/ttc2019-live-git

An NMF solution to the TTC 2019 Live Case

Georg Hinkel

Am Rathaus 4b, 65207 Wiesbaden, Germany
georg.hinkel@gmail.com

Abstract

This paper presents a solution to the BibTex to Docbook (live) case at the Transformation Tool Contest (TTC) 2019. We demonstrate how the flexible execution strategies of NMF Synchronizations can be used to obtain an incrementally maintained analysis of inconsistencies between models.

1 Introduction

In many applications, systems are modeled multiple times with various aspects in mind. Because these aspects of a system share commonalities in their understanding of a system, such as its principle structure, these models tend to have a semantic overlap. To ensure consistency, a set of rules is often used to check and enforce that these models fit to each other [1].

However, it is not feasible to resolve inconsistencies automatically, for instance if models of multiple levels of abstraction are involved: An entity present in a more abstract model often cannot be automatically added in a more detailed model as usually additional input is required. In such cases, it is necessary to temporarily allow inconsistencies between the models and to explicitly identify and manage those inconsistencies. For some inconsistencies, it is possible to resolve them automatically, for others it is not.

In the TTC 2019 BibTex to DocBook benchmark [2], the task was to identify inconsistencies between models of a bibliographic reference and a model of a book, if the latter model faces changes. Those changes are either available as mutated DocBook models or as operational change sequences describing which operations had been performed on the target model. Solutions were asked to analyze the correspondences, i.e. whether new change sequences have arisen.

In this paper, we present a solution to this benchmark using synchronization blocks and their implementation in NMF Synchronizations [3]. NMF Synchronizations allows us to obtain descriptions of change sequences straight from the specification of the consistency specification. Because inconsistencies can be repaired automatically towards the target DocBook model, the same specification can also be used to transform a BibTex model into a corresponding DocBook model. Both the transformation and also the analysis for inconsistencies can be run incrementally, i.e. the engine automatically attaches to the source and target model and notifies new inconsistencies or resolves them automatically.

2 Synchronization Blocks and NMF Synchronizations

Synchronization blocks are a formal tool to run model transformations in an incremental (and bidirectional) way [3]. They combine a slightly modified notion of lenses [4] with incrementalization systems. Model properties and methods are considered morphisms between objects of a category that are set-theoretic products of a type (a set of instances) and a global state space Ω.

In: A. Garcia-Dominguez, G. Hinkel and F. Krikava (eds.): Proceedings of the 12th Transformation Tool Contest, Eindhoven, The Netherlands, 19-07-2019, published at http://ceur-ws.org

A (single-valued) synchronization block $\boldsymbol{S}$ is an octuple $(A, B, C, D, \Phi_{A-C}, \Phi_{B-D}, f, g)$ that declares a synchronization action given a pair $(a, c) \in \Phi_{A-C} : A \cong C$ of corresponding elements in a base isomorphism Φ_{A-C}. For each such a tuple in states (ω_L, ω_R), the synchronization block specifies that the elements $(f(a, \omega_L), g \nearrow (b, \omega_R)) \in B \times D$ gained by the lenses f and g are isomorphic with regard to Φ_{B-D}.

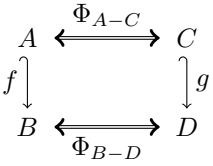


Figure 1: Schematic overview of unidirectional synchronization blocks

A schematic overview of a synchronization block is depicted in Figure 1. The usage of lenses allows this declarations to be enforced automatically and in both directions. The engine simply computes the value that the right selector should have and enforces it using the PUT operation. Similarly, a multi-valued synchronization block is a synchronization block where the lenses f and g are typed with collections of B and D, for example $f : A \hookrightarrow B*$ and $g : C \hookrightarrow D*$ where stars denote Kleene closures.

Synchronization Blocks have been implemented in NMF Synchronizations, an internal DSL hosted by C# [3], [5]. For the incrementalization, it uses the extensible incrementalization system NMF Expressions [6]. This DSL is able to lift the specification of a model transformation/synchronization in three quite orthogonal dimensions:

- **Direction:** A client may choose between transformation from left to right, right to left or in check-only mode
- **Change Propagation:** A client may choose whether changes to the input model should be propagated to the output model, also vice versa or not at all
- **Synchronization:** A client may execute the transformation in synchronization mode between a left and a right model. In that case, the engine finds differences between the models and handles them according to the given strategy (only add missing elements to either side, also delete superfluous elements on the other or full duplex synchronization)

The check-only mode has been implemented very recently and the BibTex to DocBook case has been the first actual usage. In this mode, the engine will try to match the given model and report inconsistencies. This direction is compatible with incremental execution. However, one-way change propagation is not supported[1] (must be two-way or disabled) and the synchronization is required to be bidirectional.

This flexibility makes it possible to reuse the specification of a transformation in a broad range of different use cases. Furthermore, the fact that NMF Synchronizations is an internal language means that a wide range of advantages from mainstream languages, most notably modularity and tool support, can be inherited [7].

3 Solution

When creating a model transformation with NMF Synchronizations, one has to find correspondences between input and target model elements and how they relate to each other. The first correspondence is usually clear and is the entry point for the synchronization process afterwards: The root of the input model is known to correspond to the root of the output model, in our case the `BibTeXFile` element should correspond to the `DocBook` element.

For the isomorphism from a `BibTex` file to a `DocBook`, we need to override the creation of target models. If the `DocBook` element is missing, it should be created with the required sections right from the start. For this, the initialization syntax as depicted in Listing 1 can be used.

To synchronize the contents of a BibTexFile with the contents of a DocBook, we need to specify synchronization blocks accordingly. As an example, the synchronization of the BibTex entries with the paragraphs in the references list is depicted in Figure 2. Because the DocBook does not have a direct reference to the references list, we created a helper method to identify the references list.

[1]The problem here is that the internal DSL of NMF Synchronizations allows a write-only interface for one-way synchronization blocks. Thus, an incrementally maintained list of inconsistencies can only take changes in the source model into account and could therefore produce misleading results which is why the feature has not been implemented, yet. For a bidirectional transformation, this is not a problem, because the list of inconsistencies can always be kept consistent.

```
protected override DocBook CreateRightOutput(...) {
return new DocBook() {
  Books = {
    new TTC2019.LiveContest.Metamodels.Docbook.Book() {
      Id = "book",
      Articles = {
        new TTC2019.LiveContest.Metamodels.Docbook.Article() {
          Title = "BibTeXML to DocBook",
          Sections_1 = {
            new Sect1() { Id = "se1", Title = "References List" },
            new Sect1() { Id = "se2", Title = "Authors list" },
            new Sect1() { Id = "se3", Title = "Titles List" },
            new Sect1() { Id = "se4", Title = "Journals List" }
          }
        }
      }
    }
 }
};
}
```

Listing 1: Creating the default output model for the DocBook metamodel

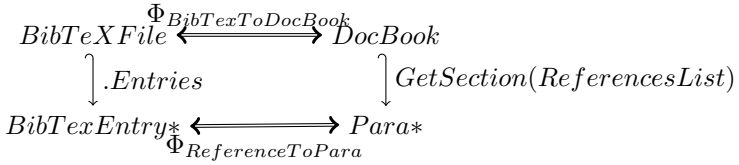


Figure 2: Synchronization block to synchronize entries with the paragraphs in the references list

The implementation for the synchronization blocks from Figure 2 is depicted in Listing 2. In particular, line 1 defines the isomorphism $\phi_{BibTexToDocBook}$, lines 3-5 implement the synchronization block from Figure 2.

NMF Synchronizations uses the incrementalization system of NMF Expressions, which allows to incrementalize also more complex queries. However, queries only have a readonly interface, which is why the C# compiler throws an error if we tried to use it for a bidirectional synchronization. To aid this problem, our solution uses a mocked class `PseudoCollection` that mimics a collection interface on top of a query but throws runtime exceptions if the collection is attempted to be modified. With this trick, the synchronization blocks for the remaining correspondence criteria can be implemented as in Listing 3.

In particular, lines 1-7 specify the synchronization of authors that appear in any of the authored entries with the paragraphs in the authors list. For this, we just iterate over all entries that are authored entries, select all their autors, do a deduplication and sort the result by the author name. Lines 9-13 specify the synchronization of titles in a very similar way. Last, lines 15-21 synchronize the journal names with the paragraphs in the corresponding section.

These synchronization blocks all reference other synchronization blocks responsible for how the paragraphs are laid out exactly. In particular, the synchronization rule `ReferenceToPara` has an instantiating rule[2] for each concrete type in the BibTex model.

Furthermore, the synchronization rules can (and in general should) specify when NMF Synchronizations should establish a correspondence link between two existing model elements. This is in particular required if a

[2]NMF Synchronizations – as NMF Transformations – supports a form of transformation rule superimposition. This concept is called transformation rule inheritance as it is realized through inheritance of the respective transformation rule or synchronization rule. In contrast, an instantiating rule is a child rule that maps a part of the input space (usually denoted through a subtype of the input type) to a part of the output space (again, usually denoted through a subtype of the output type), cf. [8].

```
public class BibTexToDocBook : SynchronizationRule<BibTeXFile, DocBook> {
    public override void DeclareSynchronization() {
       SynchronizeMany(SyncRule<ReferenceToPara>(),
         bibTex => bibTex.Entries,
         docBook => GetSection(docBook, "References List"));
    }
}
```

Listing 2: Definition of synchronization blocks from Figure 2 in NMF Synchronizations

```
SynchronizeMany(SyncRule<AuthorToPara>(),
  bibTex => new PseudoCollection<IAuthor>(
      bibTex.Entries.OfType<IAuthoredEntry>()
            .SelectMany(entry => entry.Authors)
            .Distinct()
            .OrderBy(a => a.Author_)),
  docBook => GetSection(docBook, "Authors list"));

SynchronizeMany(SyncRule<TitledEntryToPara>(),
  bibTex => new PseudoCollection<ITitledEntry>(
      bibTex.Entries.OfType<ITitledEntry>()
            .OrderBy(en => en.Title)),
  docBook => GetSection(docBook, "Titles List"));

SynchronizeMany(SyncRule<JournalNameToPara>(),
  bibTex => new PseudoCollection<string>(
      bibTex.Entries.OfType<Bibtex.IArticle>()
            .Select(article => article.Journal)
            .Distinct()
            .OrderBy(journal => journal)),
  docBook => GetSection(docBook, "Journals List"));
```

Listing 3: Synchronization blocks to synchronize the authors in the authors list, the titles in the titles list and the journals in the journals list.

```
public override bool ShouldCorrespond(IBibTeXEntry left, IPara right, .. context) {
  return right.Content != null && right.Content.StartsWith($"[{left.Id}]");
}
```

Listing 4: Specification when a paragraph should be considered corresponding to a BibTex entry

synchronization is applied to existing models.

In our case, we could establish a correspondence link in case the article starts with the Id of the entry in square brackets as implemented in Listing 4. This allows to synchronize elements with custom global identifiers.

The solution offers two ways to execute it. In the batch mode, the synchronization is run in check-only mode without change propagation, loading the reference `BibTex` model and the mutated `DocBook` model. The implementation is depicted in Listing 5.

Alternatively, the incremental version synchronizes the initial `BibTex` model with the initial `DocBook` model, but with change propagation set to two-way as depicted in Listing 6. Afterwards, the solution loads the target model changes and applies them. Because the changes are propagated as they are applied, the collection of inconsistencies is already maintained.

Besides the synchronization engine also supports a greenfield transformation (i.e. where no target model exists when the transformation is started), the solution is configured to run the initial synchronization in a brownfield setting both in incremental and batch mode. In the batch mode, only inconsistencies between the source model and existing target model are collected, in the incremental setting the synchronization engine collects inconsistencies (there are none as the source model and initial target model are consistent) and installs change propagation hooks to get notified when further changes cause new inconsistencies or resolve existing ones.

In the solution, the inconsistencies are only counted. As an alternative, the changes can be inspected to find out what these inconsistencies actually look like and offer functions to resolve them on either left or right model.

4 Evaluation

We noted that the original solution produced different results in batch or incremental mode. The reason for this was that the definition of correspondence between authors and paragraphs being that a correspondence should be established if the content of the paragraph is the name of the author. However, this is sensitive to changes that modify the contents of the paragraph. While the incremental execution is aware that the paragraph

```
var context = transformation.Synchronize(ref bibTex, ref docBook,
    SynchronizationDirection.CheckOnly,
    ChangePropagationMode.None);
Report("Run", context.Inconsistencies.Count);
```

Listing 5: Running the synchronization in batch mode

```
var context = transformation.Synchronize(ref bibTex, ref initialDocBook,
    SynchronizationDirection.CheckOnly,
    ChangePropagationMode.TwoWay);
var changes = repository.Resolve(...).RootElements[0] as ModelChangeSet;
Report("Load");
changes.Apply();
Report("Run", context.Inconsistencies.Count);
```

Listing 6: Running the synchronization incrementally

corresponds to the author and therefore reports a single inconsistency (that the content of the paragraph no longer matches the name of the author), the batch execution reports two inconsistencies: One that a paragraph has no corresponding author and another that an author has no corresponding paragraph. The reason for this is that the batch mode does not have the history and is not aware that the paragraph was corresponding to the author before.

This behavior can be easily fixed by actually using the unique identifiers that happen to exist in the metamodel in question as they can be used to identify the *same* paragraph even though its contents have changed. Doing so, also the batch execution detects that the contents of the paragraph have changed. The default behavior for matching elements is currently not taking identifiers into account, because NMF Synchronizations is independent of the model representation of NMF.

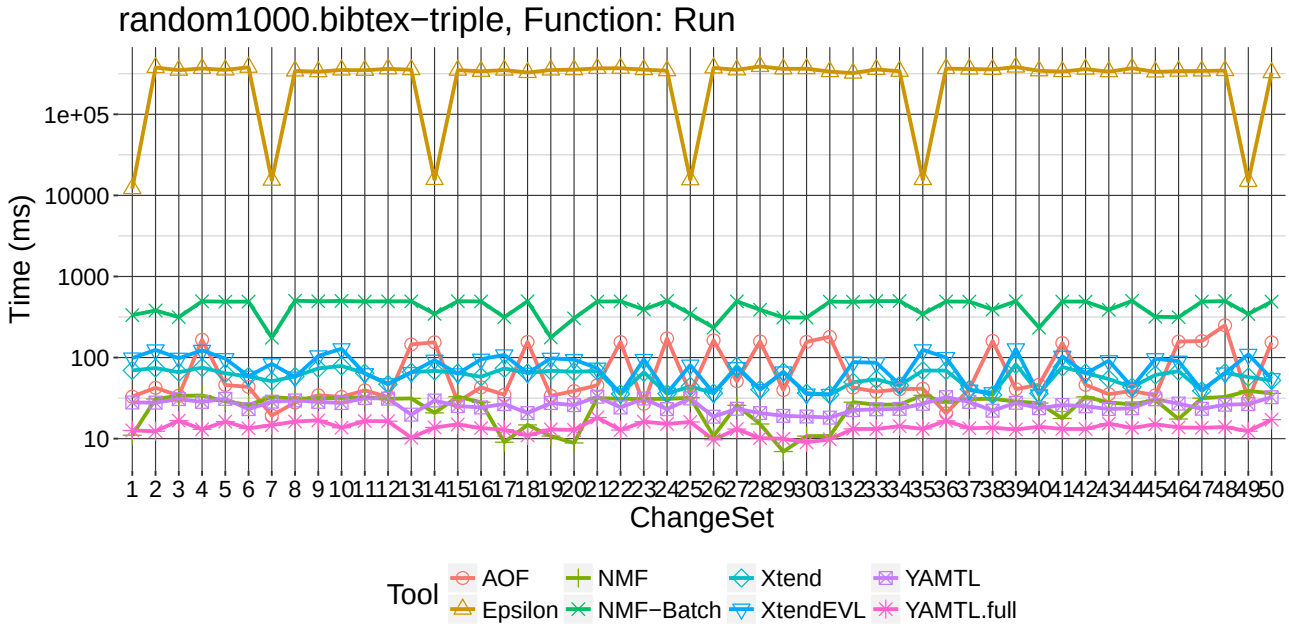


Figure 3: Performance results of the solutions at the TTC 2019

Figure 3 shows the performance results of the solutions submitted to the TTC 2019 for the largest input model provided with triple changes. These results were produced through a Google Cloud Compute Engine c1-standard-4 machine with 16GiB of RAM and 30GB SSDs, using a Docker image produced through the Dockerfile in the root of the benchmark repository. For NMF, two versions are depicted. The NMF version uses the incremental change propagation of inconsistencies. In contrast, the NMF Batch solution runs a complete consistency check on the source model and modified target model, reflecting a scenario where a description of changes is not available.

The results show that the batch version is significantly slower than competing solutions, which is clear, given that the entire modified target model has to be considered and correspondence relations have to be established. The incremental solution, however, is among the fastest solutions, indicating that it is about as fast as solutions that basically analyze the target model changes manually.

At the TTC 2019, the NMF solution was the only solution that was able to entirely use the same consistency specification for transforming the source model to the target model and to identify inconsistencies afterwards. Further, the NMF solution has won the best integration award and the first place in the audience award.

5 Conclusion

We think that the NMF solution highlights the advantages model transformations based on synchronization blocks can offer in terms of flexibility. A single specification of consistency relationships between the `BibTex` model and the `DocBook` model suffices to transform an existing `BibTex` model to a `DocBook` model or identify inconsistencies between an existing source and an existing target model. Furthermore, both the transformation and the consistency check can be run incrementally, i.e. the synchronization engine attaches to the source (or target) model and reports or resolves inconsistencies as they occur.

References

[1] M. E. Kramer, "Specification languages for preserving consistency between models of different languages," PhD thesis, Karlsruhe Institute of Technology (KIT), 2017, 278 pp.

[2] A. Garcia-Dominguez and G. Hinkel, "The TTC 2019 Live Case: BibTeX to DocBook," in *Proceedings of the 12th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2019) federation of conferences*, ser. CEUR Workshop Proceedings, CEUR-WS.org, 2019.

[3] G. Hinkel and E. Burger, "Change Propagation and Bidirectionality in Internal Transformation DSLs," *Software & Systems Modeling*, 2017.

[4] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, no. 3, 2007.

[5] G. Hinkel, "Change Propagation in an Internal Model Transformation Language," in *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*, Springer International Publishing, 2015, pp. 3–17.

[6] G. Hinkel, R. Heinrich, and R. Reussner, "An extensible approach to implicit incremental model analyses," *Software & Systems Modeling*, 2019.

[7] G. Hinkel, T. Goldschmidt, E. Burger, and R. Reussner, "Using Internal Domain-Specific Languages to inherit Tool Support and Modularity for Model Transformations," *Software & Systems Modeling*, pp. 1–27, 2017.

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

YAMTL Solution to the TTC 2019 BibtexToDocBook Case

Artur Boronat
School of Informatics, University of Leicester, UK
aboronat@le.ac.uk

Abstract

Model-to-model transformations are often used to define consistency maintainers between metamodels. When these are executed incrementally, only the part of the model transformation that is affected by model updates is executed. The *TTC2019 BibTeX2DocBook* case proposes to evaluate mechanisms for avoiding the situation where updates, to DocBook models, introduce undesirable inconsistencies that are not valid according to the transformation. In this paper, we present the application of an inconsistency specification language, available within YAMTL, for locating when a model update yields illegal changes that break the consistency relation defined by a YAMTL model-to-model transformation. Moreover, a re-implementation of the reference ATL transformation in YAMTL is presented and compared against the most efficient transformation engine for ATL.

1 Introduction

Incrementality is an important optimization technique when transforming very large models, which avoids redundant computation. A transformation engine with support for incremental forward propagation of updates executes the transformation and then, when given an update to the source model, the changes are isolated and only the part of the transformation that is sensitive to them is re-executed.

YAMTL [Bor18] is a model transformation engine for very large models that enables the execution of model transformations both in batch mode and in incremental mode, without additional user specification overhead. It is implemented in Xtend, which transpiles to Java, and uses EMF as the (meta-)modeling front end. YAMTL implements a forward delta propagation procedure [Bor19] for executing model transformations in incremental mode that can handle documented change scenarios [BRVV12], i.e. documents representing a change to a given source model. Such documents are defined with the EMF change model [SBPM09], both conceptually and implementation-wise, guaranteeing interoperability with EMF-compliant tools. EMF is de facto implementation of the MOF standard [OMG16] for meta-modeling, in particular of *essential MOF (eMOF)*.

In this article we present the YAMTL solution to the *TTC19 Bibtex to Docbook Case* [GDH19], which has two parts: a transformation of documents from Bibtex to Docbook defining a consistency relation between source documents in Bibtex and target documents in Docbook; and a number of updates on the target document, which may invalidate the consistency relation. The first requirement is optional and consists in re-implementing the reference ATL transformation; the second requirement is mandatory and consists in finding out whether the target updates affect the validity of the consistency relation of the source Bibtex models against the mutated Docbook models, following a description of invalid updates on target models. In the mandatory requirement, the check should be possible without a full re-run of the transformation and inconsistencies should be reported clearly to the user. The solution is available at `http://bit.ly/ttc19-live-yamtl`.

In: A. Garcia-Dominguez, G. Hinkel and F. Krikava (eds.): Proceedings of the 12th Transformation Tool Contest, Eidhoven, Netherlands, 19-07-2019, published at http://ceur-ws.org

In the following, the YAMTL solution, starting with the mandatory requirement, and then following with the optional one. Finally, we discuss the main strengths of the approach.

2 Solution to Mandatory Requirement

YAMTL has been extended with an inconsistency specification DSL to filter out updates that are not valid w.r.t. the consistency relation specified by the transformation. This facility is generic and can be reused for any type of consistency relation. In this section, we describe how YAMTL's inconsistency specification language has been used to check for inconsistencies in updates without having to re-execute the transformation. Inconsistency reporting is considered in the next section, on the optional requirement.

2.1 Inconsistency Specification

An inconsistency specification consists of a list of possible inconsistency types. Each inconsistency type is declared by using the class, whose instances may affected by an update, the feature that may be updated, and a list of conflictive update types. Each conflictive update type is declared with an informative error, the type of update (`ADD`, `REMOVE`, `MOVE`, `UPDATE`), and an inconsistency guard that determines whether the update is conflictive or not. The inconsistency guard is optional. When none is required, the `TRIVIAL_CHECK` should be used. The inconsistency specification DSL allows for the definition of fine-grained inconsistency types, for feature bindings.

According to the acceptable mutations provided in [GDH19], the inconsistency specification of our solution, listed below, contains the following inconsistency types:

- Within `Section`s: swapping paragraphs breaks consistency (line 10) and paragraphs cannot be deleted (line 11).
- Within `Article`s: deleting sections is not allowed (line 16).
- Within `Sect1`s: deleting sections is not allowed (line 21), and paragraphs can only be added if they do not match one of the authors, titles, or journals in its `Sect1` (lines 24-28). The last inconsistency type is captured using an inconsistency guard, which checks whether there is a paragraph in `Sect1` whose contents match the contents of the paragraph being added.

```
1 class YAMTLSolution {
2   val static DocBook = DocbookPackage.eINSTANCE
3
4   val static TRIVIAL_CHECK = [EObject eObj, Object value | true] as (EObject,Object)=>boolean
5
6   @Accessors
7   val public static inconsistencySpec = #{
8     DocBook.section -> #{ 'paras' ->
9       #{
10        'Swapping paragraph' -> (YAMTLChangeType.MOVE -> TRIVIAL_CHECK),
11        'Deleting paragraph' -> (YAMTLChangeType.REMOVE -> TRIVIAL_CHECK)
12      }
13    },
14    DocBook.article -> #{
15      'sections_1' -> #{
16        'Deleting sections' -> (YAMTLChangeType.REMOVE -> TRIVIAL_CHECK)
17      }
18    },
19    DocBook.sect1 -> #{
20      'sections_2' -> #{
21        'Deleting sections' -> (YAMTLChangeType.REMOVE -> TRIVIAL_CHECK)
22      },
23      'paras' -> #{
24        'Adding an existing paragraph to Sect1' -> (YAMTLChangeType.ADD -> [ EObject eObj, Object value |
25          val sect1 = eObj as Sect1
26          val para = value as Para
27          sect1.paras.exists[it.content.startsWith(para.content)]
28        ] as (EObject, Object)=>boolean)
29      }
30    }
31  } as Map<EClass,Map<String,Map<String,Pair<YAMTLChangeType,(EObject,Object)=>boolean>>>>
32 }
```

2.2 Admissibility Tests

The YAMTL model transformation $\mathcal{T}$ between the `BibTeX` metamodel $\mathcal{M}_{BibTeX}$ and the `DocBook` metamodel $\mathcal{M}_{DocBook}$, can be regarded as the definition of a consistency relation $\mathcal{T}(\mathcal{M}_{BibTeX}, \mathcal{M}_{DocBook})$, and the predicate $\mathcal{T} \models (m_{BibTeX}, m_{DocBook})$ denotes that a source model m_{BibTeX} is consistent with a target model $m_{DocBook}$ according to $\mathcal{T}$.

An inconsistency specification defined over the target metamodel, $\mathcal{I}(\mathcal{M}_{DocBook})$, as presented in the previous section, defines the type of inconsistencies that are not allowed for target models. A target model update δ_t is admissible, $\mathcal{T} \models_{\mathcal{I}} \delta_t$, if and only if $\mathcal{I}(\mathcal{M}_{DocBook}) \not\models \delta_t$. That is, when it does not yield any inconsistency. In YAMTL, $\mathcal{T} \models_{\mathcal{I}} \delta_t$ is implemented with the expression

```
xform.admissibleChange(model, delta, iSpec)
```

which checks whether the `delta` for `model` is valid according to the inconsistency specification `iSpec` for the transformation `xform`. This is the operation that has been used to report whether a model update is consistent or not in the solution and it does not require the transformation `xform` to be re-executed.

3 Solution to Optional Requirement

The reference ATL transformation used to transform documents from Bibtex to Docbook was improved[1] by using rules that capture the four possible combinations of `title` and `journal` in `BibTeXEntry` objects using multiple rule inheritance in EMFTVM [WTCJ11]. EMFTVM matches those rules using a layered algorithm that uses a network of model constraints such that rules whose parent rules' matching conditions are not satisfied are not eligible to be matched, thus skipping rules during the matching phase.

In this section, the performance of the YAMTL implementation of the improved ATL transformation has been empirically evaluated using the TTC benchmark harness and compared against the analogous ATL transformation on EMFTVM. The traceability support built in the YAMTL transformation engine has been used to report inconsistencies according to the inconsistency specification presented in the previous section.

3.1 Transformation Re-Implementation in YAMTL

YAMTL also supports multiple rule inheritance [Bor18], benefitting from a thrifty application of rules as explained above, and the improved ATL transformation has been mirrored in an improved YAMTL transformation.[2] The results of both transformations have been checked for correctness and their outputs are identical but for the randomly generated identifiers. For obtaining performance results, we have executed the benchmark for models corresponding to size factors up to 10,000, for one single `mutantSet` and for one single `mutant`, as transformations are executed in batch mode, without propagation of updates. In addition, one thousand iterations were run for each input model and the results correspond to cold run times. Table 1 shows mean run times (in *ms.* unless stated otherwise) used by both tools. YAMTL takes about a third of the time used by EMFTVM as the size factor increases.

Size factor	ATL (EMFTVM)	YAMTL
10	352	84
100	456	143
1,000	766	297
10,000	3.6 s.	1.3 s.

Table 1: Model element cardinalities and run times in ms. (unless stated otherwise)

3.2 Reporting: Finding All Inconsistencies

The tool also provides the operation

```
xform.findInconsistenciesInChange(model, delta, iSpec, enableReport),
```

[1] `http://bit.ly/ttc19live-emftvm`

[2] The full solution included the original YAMTL transformation and the improved one was produced after the contest.

which finds all of the inconsistencies that are found in $\mathcal{T} \not\models_{\mathcal{I}} \delta_t$. The signature of the operation is the same as for `admissibleChange` but, in addition, it provides a boolean flag `enableReport`. If reporting is enabled, a graphical report is generated with all of the found inconsistencies, as shown in Fig. 1. For each inconsistency, the report includes the reason for the inconsistency (stating the description of the inconsistency, the object affected, the feature that was updated, the type of update and the value that was affected) and the transformation step that has been invalidated by the target model update (stating the rule that was applied, the excerpt of the source model m_{BibTeX} involved in the rule application, the excerpt of the target model $m_{DocBook}$ produced by the rule before the update, and the same excerpt of the target model after the update, i.e. $\delta_t(m_{DocBook})$). Graphical representation of models are obtained using PlantUML.

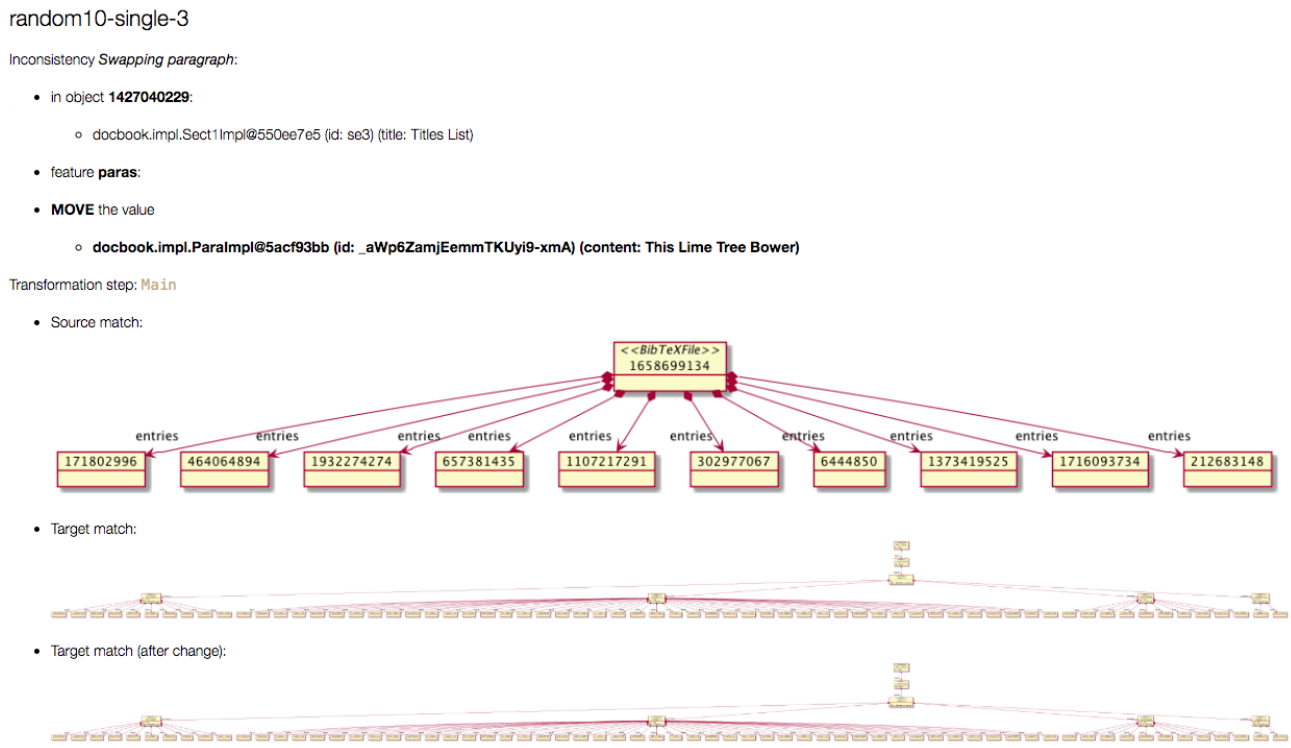


Figure 1: Outline of Solution

When reporting is enabled, the transformation `xform` must be executed by YAMTL in advance in order to provide traceability information from source to target models. Although the reporting feature is experimental, it serves as a proof-of-concept of the possibilities that YAMTL offers when informing modellers about inconsistencies that are created by target model updates, providing traceability to the affected part of the source model.

4 Conclusions

In this paper, we have discussed a full solution to the TTC'19 live case, including the use of YAMTL's inconsistency specification language for identifying inconsistencies in model updates (mandatory task) and a re-implementation of the ATL transformation (optional task) in YAMTL, which facilitates reporting inconsistencies when model updates are applied to the generated target model. Our solution provides the following strengths:

Granularity of Inconsistency Detection. Inconsistency specifications enable the identification of atomic model updates that lead to inconsistencies at binding level, including the possibility of defining complex checks that traverse the affected model using inconsistency guards. Inconsistencies in composite model updates are analysed by decomposing the model updates into atomic ones.

Reuse. Inconsistency specifications are defined at the metamodel level and the specification language itself is metamodel agnostic. Therefore it can be used for defining the language of admissible model updates for any consistency maintainer that defines a consistency relation between metamodels.

Reporting. The graphical reports generated by YAMTL enumerate all of the possible inconsistencies in model updates. YAMTL exploits the traceability links built during a transformation execution in order to illustrate the impact of a model update on a target model in the source model.

Performance. The goal of the optional part of the case consisted in providing an improvement of the original model transformation. The YAMTL transformation is analogous to the improved ATL transformation using multiple rule inheritance, and it has been shown that YAMTL executes the transformation in about one third of the time that EMFTVM uses to run the improved ATL transformation.

Some of the YAMTL features used in this case are experimental and their refinement is reserved for future work.

References

[Bor18] Artur Boronat. Expressive and efficient model transformation with an internal DSL of Xtend. In *MODELS 2018*, pages 78–88. ACM, 2018.

[Bor19] Artur Boronat. Offline delta-driven model transformation with dependency injection. In *FASE 2019*, volume 11424 of *LNCS*, pages 134–150. Springer, 2019.

[BRVV12] Gábor Bergmann, István Ráth, Gergely Varró, and Dániel Varró. Change-driven model transformations - change (in) the rule to rule the change. *Software and System Modeling*, 11(3):431–461, 2012.

[GDH19] Antonio García-Dominguez and Georg Hinkel. The TTC 2019 Bibtex to Docbook Case. In Antonio García-Dominguez and Georg Hinkel, editors, *Proceedings of the 12th Transformation Tool Contest (TTC@STAF)*, CEUR Workshop Proceedings. CEUR-WS.org, July 2019.

[OMG16] OMG. Meta Object Facility (MOF) 2.5.1 Core Specification, 2016.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[WTCJ11] Dennis Wagelaar, Massimo Tisi, Jordi Cabot, and Frédéric Jouault. Towards a General Composition Semantics for Rule-Based Model Transformation. In *MoDELS*, volume 6981, pages 623–637. LNCS, 2011.