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 or diagram 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 a detailed transformation algorithm that creates a binary decision tree or diagram. The implementation contains one solution that creates and ordered BDT or BDD and one that creates an unordered BDD or BDT, while the unordered one works on a special port selection criteria. We evaluate our RSYNC transformation approach and show the advantages of the role concept in such a transformation.

1 Introduction

The Transformation Tool Contest (TTC) describes every year 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 tree. The presented case [GD19] provides two different target metamodels to choose from. These metamodels differ in their representation from a tree structure (BDT) to a graph structure (BDD). To validate the overall solution, the target models only must correspond to the source model and there are no requirements regarding the order of the nodes or the optimality of the target model. To solve this problem, we use the RSYNC [WSK+18] approach, which describes rules for creating, deleting, modifying, and importing elements in the source model based on the role concept and can apply them incrementally. In addition, we present the usability of the RSYNC approach for the TTC case and automatically set up explicit traceability links between the source and target models. These links allow the modification of the source model at runtime with automatic consistency preserving mechanisms.

The remainder of this paper is structured as follows. 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.

The implementation can be found in a public Git repository[1].

2 Background

The role-based synchronization approach (RSYNC) [WSK+18] used in this paper is based on the role concept known from the 70s. In the 2000s, Steinmann and Kühn *et al.* [Ste00, KLG+14] identified 27 features that describe the nature of roles in terms of their behaviour, their relational dependence to each other, and their

[1] https://git-st.inf.tu-dresden.de/ttc/bdd

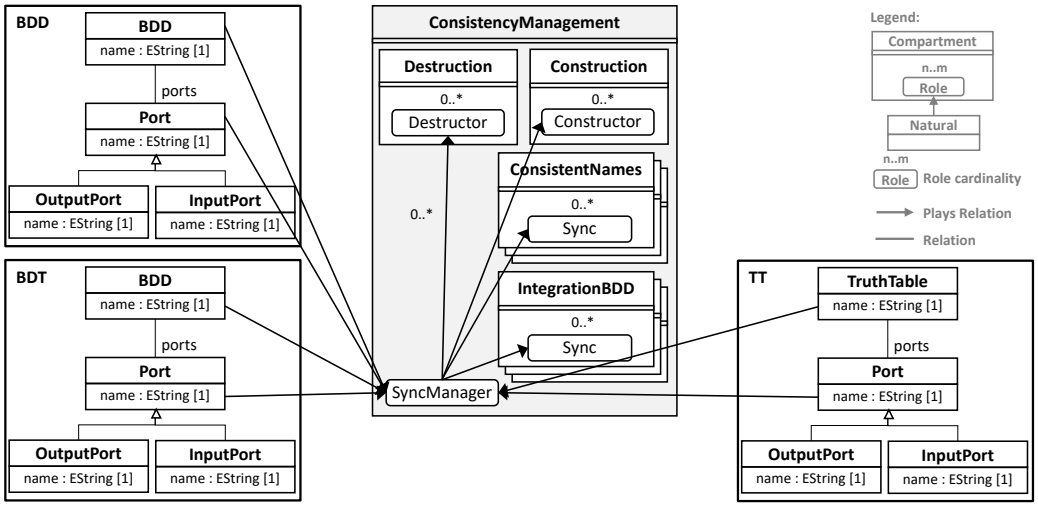


Figure 1: Role-based synchronization approach [WSK+18].

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 runtime adaptation and evolution. 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 transformations and for permanent consistency preservation of several models.

The RSYNC approach uses the advantages of roles and describes an approach for synchronizing multiple related models. In Figure 1, the concept is visualized on metamodel level using a partial model of the TTtoBDD case. The core of the approach is the `ConsistencyManagement` compartment, which manages all rules of the transformation and takes care of the execution. In the role concept, compartments represent a kind of context in which roles exist and interact with each other. The RSYNC approach distinguishes 4 types of rules, each of which is modelled in their own compartments: (1) *Creation 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) *Deletion rules* describe how to deal with the deletion of elements. (3) *Change rules* indicate how to deal with attribute or reference changes in connected models. (4) At least, *integration rules* describe rules on how to create a completely new model from an existing one. These integration rules describe in the TTC case the main transformation step in which a new BDD or BDT model is created from an existing TT model. For incremental changes, rules of the other types have also been 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 some roles in the rule compartment of the different types and react with it to changes without knowing the connection to a synchronization. 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

In this section, we describe the construction of a binary decision tree (BDT) and diagram (BDD) from a truth table (TT). Since the first version of the TTC only included creating a BDT, this is most intensively discussed in this publication. In addition, our algorithm with a simple heuristic creates an unordered BDT or BDD, i.e., the order of the input ports may differ between the two subtrees of subtree.

3.1 Computing a Disordered Binary Decision Trees

For the creation of a BDT, the simplest possibility would be to define a fixed input port order and to generate the subtrees and child nodes based on this. We used such an algorithm for a simple ordered implementation, that selects only the next node for a new subtree. Since this mechanism creates a full tree in the worst case, we decided to use another method, which does not guarantee the optimality of the tree but should create better results. The steps that the algorithm goes through are described below.

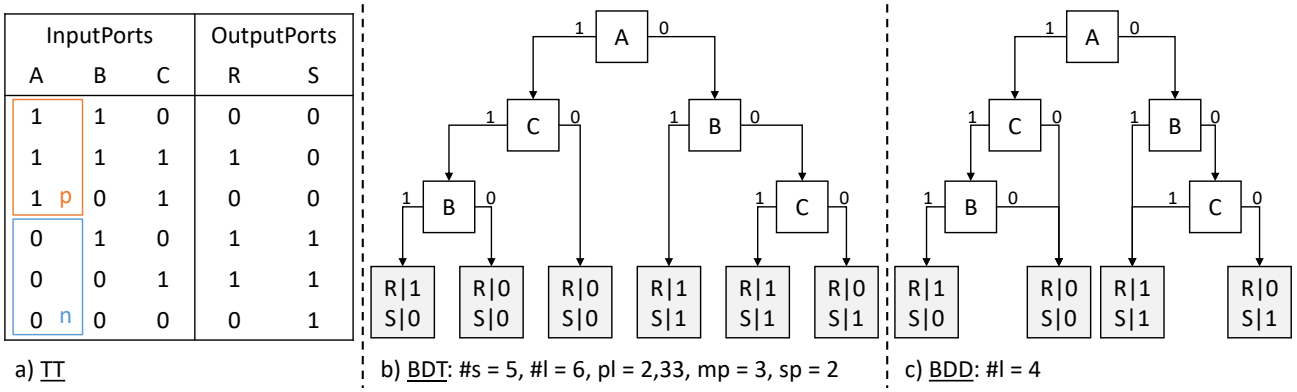


Figure 2: Example input and output model results for TT, BDT, and BDD model.

1. A new `BDD` element with the same name is created for each `TruthTable` element.
2. For each input and output port of the `TruthTable` element, a new input or output port is created in the target model and linked to the `BDD` element.
3. For each input port, the selected rows (all at the beginning) are separated into a list of *true* (rows where cells of input port have value 1) and *false* (rows where cells of input port have value 0) rows. Each list now counts how many variants of output ports and their values are contained. In Figure 2a, the variants for `InputPort` A are (`OutputPort`: (R/1, S/0), (R/0, S/0)) in *true* rows and (OutputPort: (R/1, S/1), (R/0, S/1)) in *false* rows. The sum of these variants must be minimal for an input port to be selected as the next subtree. This is the case for `InportPort` A in Figure 2a.
4. Step 3 is now running recursively which minimizes the number of lines in each level. If only one variant for a set of rows in a subtree is available, Step 5 is executed for these rows.
5. Creates a Leaf node with the corresponding assignments.

The result of the fifth step is a BDT as shown in Figure 2b. When creating subtrees and leaves, the *play* relations in the RSYNC approach create links between these elements and their rows from the TT model. This makes it possible to track which subtree are traced from which rows from the TT.

3.2 Creating Disordered Binary Decision Diagram from Unordered Binary Decision Trees

The algorithm used here corresponds in large part to the algorithm used for the BDTs in the previous paragraph. The only difference is that the leaf nodes are combined at the end to one if they would contain the same assignments. Figure 2c shows an example of this. Thus, the algorithm differs in Step 5, where only one leaf is created, if no identical leaf is already created and used somewhere.

3.3 Quality Metrics of the BDD and BDT Results

The created BDT and BDD must be correct in comparison to the entered TT. Since the employed algorithm already ensures this, the correctness is not considered as a metric. Metrics are considered as:

- Number of decision nodes (#s): Number of all subtrees without the leaves.
- Number of assignment nodes (#l): Number of all created leaves.
- Average path length (pl): Computes the average path length of the BDD or BDT.
- Longest path (mp): Longest of the created paths usually corresponds to the number of input ports.
- Shortest path (sp): Shortest path to reach a leaf.

The results of these metrics are given for our example in Figure 2. If there is always a single best choice for selecting an input port in the subtree decision, the resulting BDT and BDD from our algorithm only differ in the number of leaf nodes, since only step 5 has differs. Otherwise, the selection decision is non-deterministic, potentially resulting in different port orders for the BDT and BDD, because the used *Scala Sets* are always unordered.

4 The Transformation Toolchain

This section contains a step-by-step description of all necessary transformation processes within the RSYNC approach. The workflow starts with (1) reading the metamodel, continues with (2) generating adapted classes

for role-based programming with SCROLL, (3) reading the TT model and (4) transforming this source model into one instance of the two target metamodels. In the last step (5), the result is written into a model file.

4.1 Reading the Metamodels and Generating a Specific Role Model

For the first two steps, there is a generator, which reads metamodels and generates adapted Scala code, which is then used for the integration into the RSYNC environment. These newly generated Scala classes are necessary to automatically detect and propagate incremental changes from the source model to the target model. The generated Scala classes can be viewed in the packages *sync.bdd*, *sync.bddg*, and *sync.tt*. If only the transformation is performed, most of the changes from the generator are not necessary.

4.2 Reading Truth Table

The third step involves reading the truth table model and creating instances of the generated Scala classes. This step uses EMF in Scala, where the meta and instance model are loaded and all instances in Scala and references between these instances are created. Since it is possible to integrate Java libraries in Scala, this was the easiest and fastest way to integrate them in our approach.

4.3 Transforming the TT to a BDT and BDD

This step includes the algorithm as already described in Section 3. It is completely described within an integration rule and is executed automatically as soon as it is integrated in the `ConsistencyManagement` compartment. At this point, there is no domain-specific language (DSL) to describe these rules. So, they just must be implemented by hand in Scala. However, such a DSL must be turing-complete to cover all cases. We are currently developing a simple language which should describe simple transformations easily, but for more complex algorithms Scala code must be written by hand. In the classes `BdtSyncIntegration`, `BddSyncIntegration`, `BdtSyncIntegrationWithoutOrder` and `BddSyncIntegrationWithoutOrder` the current algorithms are implemented.

4.4 Printing the Result

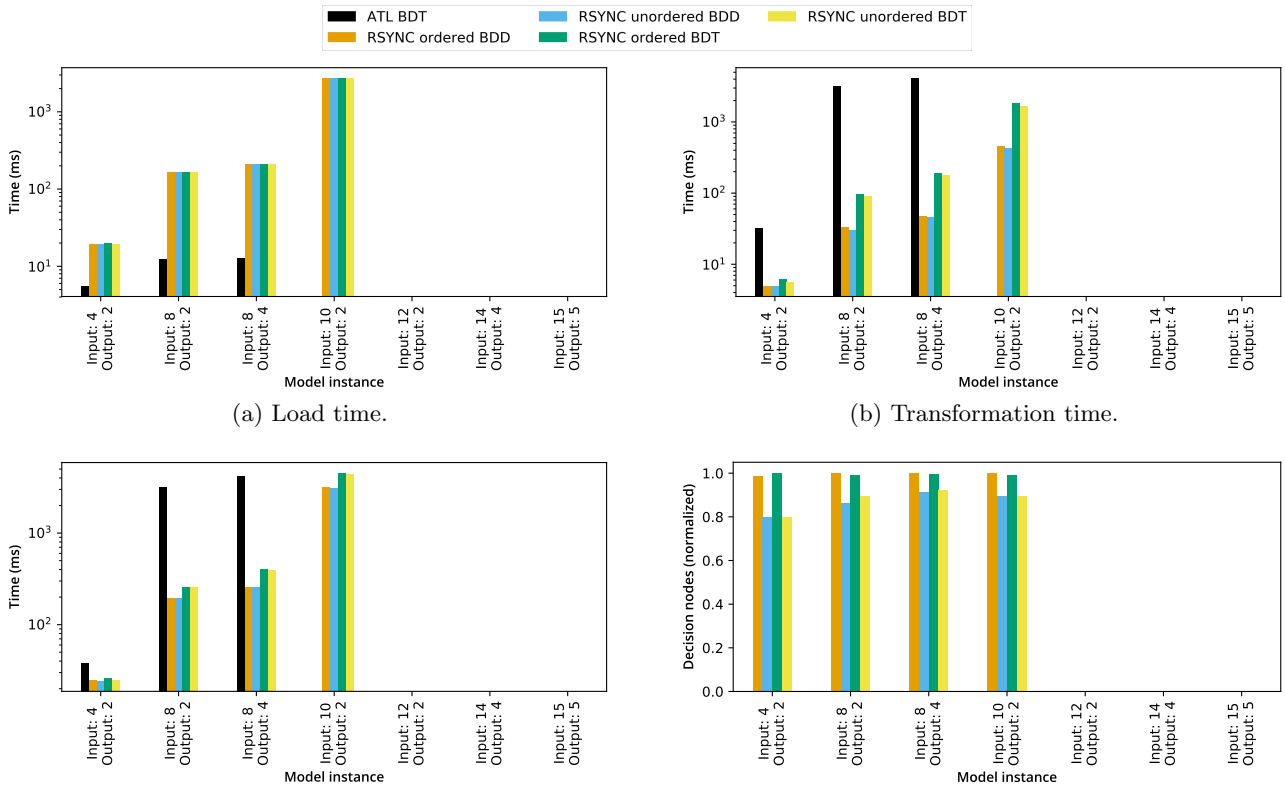
To write out the model, the generated classes from the TTC repository are included, since writing models is not part of the RSYNC application. This means that all elements are copied once and then EMF is used to write out the instance model.

4.5 Synchronization Mechanism

For the incremental synchronization of the models, there are different cases that are interesting. As already described in the algorithm, the simplest changes are creating a `TruthTable`, `InpurtPort`, or `OutputPort`, or changing names of these elements. However, we would like to discuss three interesting changes that can be made to the base model at runtime. On the one hand, it would be possible to delete an `InputPort` which would result in a completely new creation of the diagram. On the other hand, deleting an `OutputPort` would only lead to deleting assignments, but could leave an unnecessarily complex BDD as a result. The most interesting operations are inserting and deleting a complete row. When inserting a row, we have chosen a simple synchronization mechanism, so the tree is traversed and if necessary, a new subtree is inserted before a leaf node. When deleting a row, it can happen that no change at all takes place in the tree which is the case if the line is not the only one mapped to a leaf node. By using the explicit traceability links, it is possible to directly find the places where changes must be made. We have currently only implemented some consistency rules, because some changes in the target or source model do not make sense, like the deletion of subtrees in the BDD or cells in the TT.

4.6 Integration in the TTC benchmark environment

In order to deploy our solution in the provided environment for benchmarking and validation, some minor adjustments were necessary. To foster an independent evolution of both the TTC environment and our solution, we decided against editing the transformation process in a way that it might be invoked by the benchmark directly. Instead, a generic wrapper was introduced. Its main purpose was to hide the concrete process implementation and provide a generic interface to it. As the main communication means between TTC infrastructure and solution are direct process invocation and environment variables, this approach turned out to be rather straightforward:



(a) Load time.

(b) Transformation time.

(c) Total time.

(d) Number of decision nodes (normalized).

Figure 3: Evaluation result.

instead of starting our process directly, the runtime only invokes the wrapper, which is then going to unmarshall the current benchmark configuration and set-up our application accordingly. This includes determining which model to transform as well as which transformation strategy to apply (BDD or BDT, with or without maintaining input port order). Afterwards it monitors the general transformation progress and notifies other components, such as benchmark and report utility as appropriate.

5 Evaluation

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

5.1 Properties of the Transformation

By applying our role-based programming infrastructure and support tools, the provided ecore models may be reused directly. Thus, close to no boilerplate code was necessary. Furthermore, due to the unidirectional transformation, all necessary transformation rules only have to be implemented in the integration compartment. As mentioned earlier this is currently done manually but will be subject to automation in the future.

The integration compartment adheres to an abstract interface which enables an easy adaptation of the transformation process. Different strategies might be used by simply swapping the concrete implementation of the compartment (or parts of it). This approach was utilized for creating the BDTs or BDDs in an ordered or unordered variant.

5.2 Benchmark results

To evaluate the runtime performance, we used the provided benchmark script with some minor convenience modifications. We decided to run five iterations with a timeout of ten minutes per model to reduce the benchmark time. The benchmark was executed on an Intel i7-8700 workstation with 64 gigabytes of memory using Fedora Linux 29 (kernel 4.18), and OpenJDK version 1.8.

In addition to the suggested time measurement, we added some more quality metrics as discussed in Section 3.3. These include the number of decision and assignment nodes and the minimum, maximum, and average path length through the resulting diagrams.

We decided against measuring 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 objects allocated (that is other than the objects of source and target models). Here, we currently only need one extra object (`SyncManager`) for each object which acts as the traceability role. When the use case becomes more dynamic, i.e. truth tables being created or modified during runtime, this number will increase in minimal manner.

While the intelligent choosing of the next considered input port does not lead to tremendous improvements of the generated decision structure, one can see that with an increase of the source model sizes, the number of nodes as well as the average path lengths keep decreasing. However, this result is to be expected as no optimizations on the generated tree (or diagram) are performed. Such optimizations would in fact lead to much smaller structures and could easily be added as a final step.

We present the results of the runtime benchmark in Figure 3. The time necessary to load the TT model keeps increasing exponentially which is due to the increase of its size. However, we face some additional overhead and scalability issues compared to ATL, especially caused by the role concept with creating and binding roles in the SCROLL framework.

The transformation itself happens rather quickly compared to ATL and the loading time of this approach. The *"Input 12 Output 2"* model (as well as the more complex ones) may not be transformed within the given time boundaries. In contrary to ATL, our approach does not run into the time constrained with the *"Input 10 Output 2"* model visualized in Figure 3c. However, it is up to further research whether only the load phase acts as a bottleneck or whether both load phase and transformation phase are too time-consuming to fit in the time frame. Also, 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 both transformation time and the size of the resulting decision structure. In both cases the heuristic leads to slightly better results. Whereas this improvement is negligible when considering transformation time, the number of decision nodes is reduced by approximately 10% with a 20% reduction for the first model.

As the initialization time is constant, in our case we do not consider it any further here. The same applies to the total runtime which simply constitutes the sum of the runtime for each phase.

6 Conclusion and Future Work

We have shown how to apply role-based synchronization techniques to the problem of transforming truth tables to binary decision diagrams. 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 than boiled down to implementing an integration compartment to perform the actual 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 to perform role binding in a more 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

[GD19] Antonio García-Domínguez. The TTC 2019 TT2BDD Case. In *Proceedings of the 12th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2019)*, CEUR Workshop Proceedings. CEUR-WS.org, 2019.

[KBRA16] Thomas Kühn, Kay Bierzynski, Sebastian Richly, and Uwe Aßmann. Framed: Full-fledge role modeling editor (tool demo). In *Proceedings of the 2016 ACM 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 *Proceedings of the 2015 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] C. Werner, H. Schön, T. Kühn, S. Götz, and U. Aßmann. Role-based runtime model synchronization. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 306–313, Aug 2018.