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.

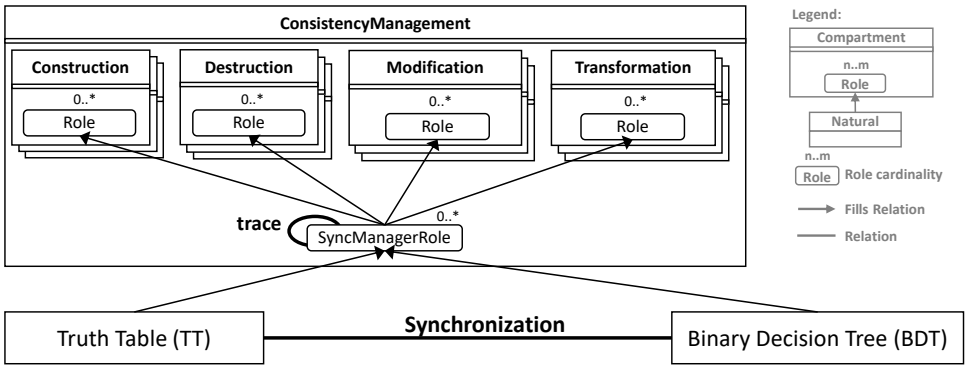


Figure 1: Role-based synchronization approach (RSYNC).

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

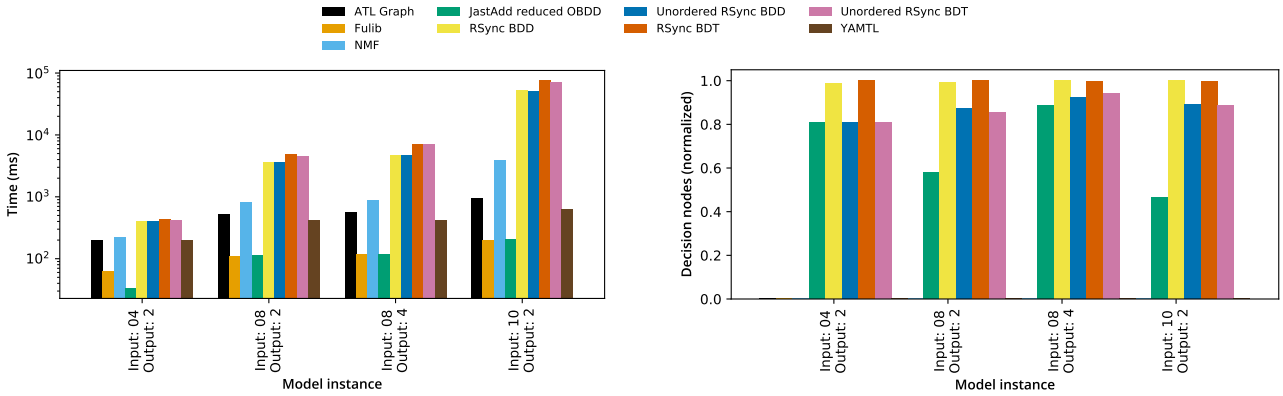
This section describes the necessary process steps to implement the transformation with the RSYNC approach. The complete approach is implemented in Scala and is based on the role-based programming language SCROLL [LA15]. For this reason, the source and target models must be available as Scala classes 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.