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}.

$$\begin{tikzcd}
A \arrow[r, Leftrightarrow, "\Phi_{A-C}"] \arrow[d, hook, "f"'] & C \arrow[d, hook, "g"] \\
B \arrow[r, Leftrightarrow, "\Phi_{B-D}"'] & D
\end{tikzcd}$$

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`

$$\begin{tikzcd}
TruthTable \arrow[r, leftrightarrow, "\Phi_{TruthTables2BinaryDecisionDiagrams}"] \arrow[d, ".Ports"] & BDD \arrow[d, ".Ports"] \\
Port* \arrow[r, leftrightarrow, "\Phi_{Port2Port}"'] & Port*
\end{tikzcd}
\qquad
\begin{tikzcd}
TruthTable \arrow[r, leftrightarrow, "\Phi_{TruthTables2BinaryDecisionDiagrams}"] \arrow[d, ".Name"] & BDD \arrow[d, ".Name"] \\
String \arrow[r, leftrightarrow, "Id_{String}"'] & String
\end{tikzcd}$$

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.

```
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.

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

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

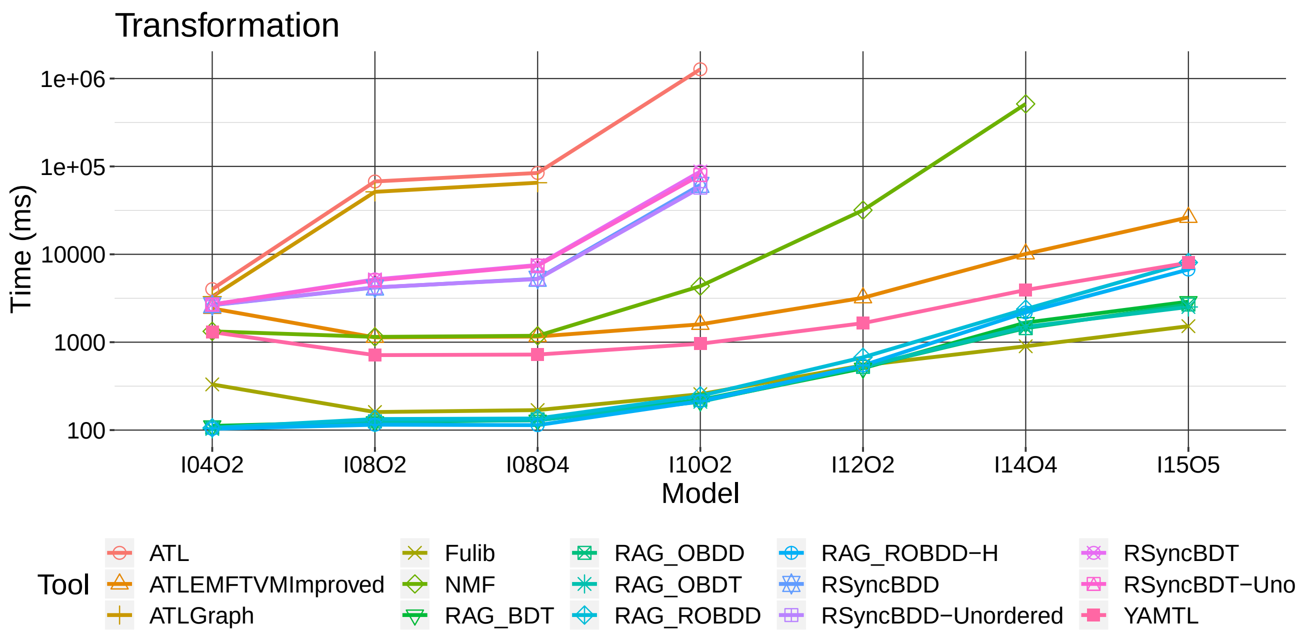

Figure 3: Performance results of the solutions at the TTC 2019

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.