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 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* [1], [2], an extensible model transformation and synchronization language and system part of the .NET Modeling Framework (*NMF*, [3], [4]). This papes introduces the solution to the TT2BDD case using NMF, in particular NMF Synchronizations. The code for the solution is available online[1].

The remainder of this paper is structured as follows: Section 2 gives a brief overview on NMF Synchronizations. Section 3 describes our solution. Section 4 reflects on the solution.

2 Synchronization Blocks and NMF Synchronizations

Synchronization blocks are a formal tool to run model transformations in an incremental (and bidirectional) way [1]. 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 (well-behaved) in-model lens $l : A \hookrightarrow B$ between types A and B consists of a side-effect free GET morphism $l \nearrow \in Mor(A, B)$ (that does not change the global state) and a morphism $l \searrow \in Mor(A \times B, A)$ called the PUT function that satisfy the following conditions for all $a \in A, b \in B$ and $\omega \in \Omega$:

$$\begin{aligned} l \searrow (a, l \nearrow (a)) &= (a, \omega) \\ l \nearrow (l \searrow (a, b, \omega)) &= (b, \tilde{\omega}) \quad \text{for some } \tilde{\omega} \in \Omega. \end{aligned}$$

The first condition is a direct translation of the original PUTGET law. Meanwhile, the second line is a bit weaker than the original GETPUT because the global state may have changed. In particular, we allow the PUT function to change the global state.

An unidirectional (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 function f and the lens g are in the dependent isomorphism Φ_{B-D}.

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

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 this declarations to be enforced automatically[2]. The engine simply computes the value that the right selector should have and enforces it using the PUT operation.

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# [1], [6]. For the incrementalization, it uses the extensible incrementalization system NMF Expressions [7]. 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 [8].

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

[2] If f was also a lens, then the synchronization block can be enforced in both directions.

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

Figure 2: Synchronization block to synchronize ports and names

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 should correspond to the root of the output model, in our case the `TruthTable` element should correspond to the `BDD` element, and their names should match. Furthermore, there is an obvious 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 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>(),
5              tt => tt.Ports,
6              bdd => bdd.Ports);
7      }
8  }
```

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 lines 4-6 implement 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.

```
1  internal class BDDLeafCollection : CustomCollection<ILeaf> {
2    private readonly BDD _bdd;
3
4    public BDDLeafCollection(BDD bdd) : base(bdd.Descendants().OfType<ILeaf>()) {
5      _bdd = bdd;
6    }
7
8    public override void Add(ILeaf item) {
9     ...
10   }
11
12   public override void Clear() {
13     _bdd.Tree = null;
14   }
15
16   public override bool Remove(ILeaf item) {
17     item.Delete();
18     return true;
19   }
20 }
```

Listing 2: Virtual collection of the leafs of a binary decision diagram

The implementation of this virtual collection is sketched in Listing 2. Note that in line 4, 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. In this case, a notification wil be issued whenever a new `Leaf` element is added somewhere in the binary decision diagram.

With this virtual collection, we can synchronize the rows of the truth table with the leafs of the binary decision diagram as in Listing 3.

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

Listing 3: Synchronizing the rows of the truth table with the leafs of the binary decision diagram

For such a corresponding pair of a row and a leaf, we need to ensure that the values for all input ports match and similarly for the output ports. For the output ports, this is rather easy because the way how this information is represented is very similar in both models: the elements have child elements that have a reference to both the output port and the value.

```
SynchronizeMany(SyncRule<OutputCell2Assignment>(),
     row => row.Cells.Where(cell => cell.Port is Metamodels.TruthTables.TT.IOutputPort),
     leaf => leaf.Assignments);
```

Listing 4: Synchronizing the output port assignments of rows and leafs

The implementation is depicted in Listing 4. We do not need to create a virtual collection for the output cells this time because NMF Expressions is able to add, remove and clear a filtered collection itself.

Finally, we need to create a virtual collection for the input assignments of a leaf in a binary decision diagram. An implementation is sketched in Listing 5.

```
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 5: Virtual colection of input port assignments of a leaf from a binary decision diagram

Again, we pass a query expression to the base class to allow NMF Expressions to incrementalize it and thus obtain notifications when the input port assignments of a leaf element change. We use the `AncestorTree` method for this that denotes the ancestors of the given element along the *Parent* relation into a collection that includes the respective ancestor and its direct child towards the given element. Further, note that the class `TreeAssignment` is just a helper class as NMF Synchronizations is not restricted to synchronizing model elements.

The `AncestorTree` method has been added to NMF specifically for this case. However, we think it can be valuable for many scenarios where tree structures have to be analyzed. Because NMF Expressions is implemented as an open framework, any users can write such extensions. In fact, the `AncestorTree` is also implemented in the *Models* sub-project of NMF and not in Expressions as the latter is independent of the model representation in NMF.

The last missing part of the implementation is the actual implementation of 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. Our implementation is currently restricted to this use case, so it would have problems when changes appear that change the input assignments of a leaf that is already part of a binary decision diagram. Right now, the implementation is relatively simple, for any assignment, a `SubTree` element is added with the according input port, adding the topmost ancestor in either the *TreeForOne* or *TreeForZero* reference. Note that most of the inner nodes created here are only temporary for the synchronization and will be deleted during the transformation.

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. 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.

```
ITree current = item;
var assignments = new Dictionary<IInputPort, bool>();
var treeStack = new Stack<ISubtree>();
while (current != null) {
  if (current.OwnerSubtreeForOne != null) {
    assignments.Add(current.OwnerSubtreeForOne.Port, true);
    treeStack.Push(current.OwnerSubtreeForOne);
    current = current.OwnerSubtreeForOne;
  }
  else if (current.OwnerSubtreeForZero != null) {
    assignments.Add(current.OwnerSubtreeForZero.Port, false);
    treeStack.Push(current.OwnerSubtreeForZero);
    current = current.OwnerSubtreeForZero;
  }
  else {
    if (_bdd.Tree == null) {
      _bdd.Tree = current;
      return;
    }
    break;
  }
}
```

Listing 6: Collecting assignments of a leaf

In a first step, we collect the assignments for a leaf to ports. This is depicted in Listing 6. In case the decision diagram does not have any tree yet, we simply set that tree and return. Otherwise, we collect the assignments in a dictionary and further keep a stack of inner tree nodes. Afterwards, 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.

If we encounter this situation, there are two possibilities, either the current path in the binary decision diagram yields a null reference or we reach an inner node referencing an input port that is not assigned anything in the input model. In the former case, we simply add the missing inner nodes. In case the order of variables is the same as in the temporary inner nodes that come with the leaf, we can save recreating those inner nodes through the stack of inner nodes. For this, we pop the topmost element from the stack if there is one, provided it references the same input port. If this is not the case, we forget the stack and create fresh inner nodes afterwards.

The more involved scenario is if we encounter an inner node for which there is no assignment in the leaf to be added. In that case, we create a new decision node for some yet unprocessed input port. If the value for this input port equals the value assigned for the current leaf, that leaf will be chosen. Otherwise, we would go on with the inner node currently visited.

```
var ownerForZero = subTree.OwnerSubtreeForZero;
var ownerForOne = subTree.OwnerSubtreeForOne;
var ownerBdd = subTree.OwnerBDD;
var peek = treeStack.Peek();
if (peek.TreeForZero == null) {
    peek.TreeForZero = subTree;
}
else if (peek.TreeForOne == null) {
    peek.TreeForOne = subTree;
}
// we need to be careful not to accidently delete peek
if (ownerForOne != null) {
    peek.OwnerSubtreeForOne = ownerForOne;
}
else if (ownerForZero != null) {
    peek.OwnerSubtreeForZero = ownerForZero;
}
else if (ownerBdd != null) {
    peek.OwnerBDD = ownerBdd;
}
return;
```

Listing 7: Inserting a new dominating inner node to the current node `subTree`

The implementation is depicted in Listing 7. We first set the reference not yet set in the selected inner node. Then, we add this inner node where the previous inner node `subTree` has been in the containment hierarchy.

The fact that NMF automatically enforces bidirectional references and ensures referential integrity has a nitpick in this situation: 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 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, which is the primary reason for the check statements in lines 11-20.

4 Reflection

Processing a tree structure incrementally is something rarely done with NMF, yet, so this case has been very interesting. In applications we have seen so far, it was usually sufficient to look at the nodes of a tree rather than the connecting edges. In this case, the edges became very important, also for the analysis from an incrementalization point of view.

The solution greatly shows the powers, but also the limitations of a declarative approach such as NMF Synchronizations.

References

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

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

[3] G. Hinkel, "NMF: A Modeling Framework for the .NET Platform," Karlsruhe Institute of Technology, Tech. Rep., 2016.

[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, "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.

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

[8] 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.