CEUR-WS.org/Vol-2550/paper5.pdf

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:

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

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.

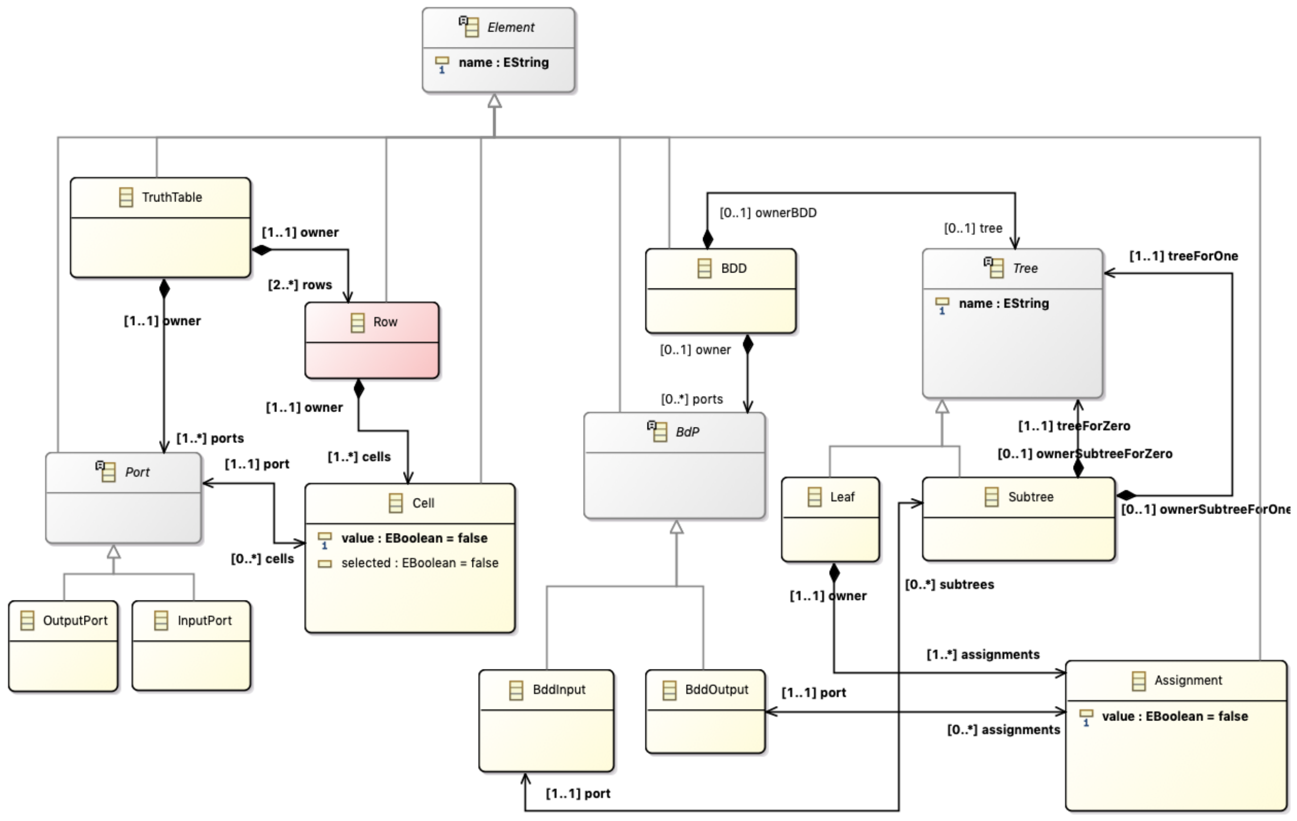


Figure 1: Merging meta-models

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.* $tree\_owner \in Tree \rightarrowtail\!\!\!\!\!\!\!+\;\; BDD$). 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.

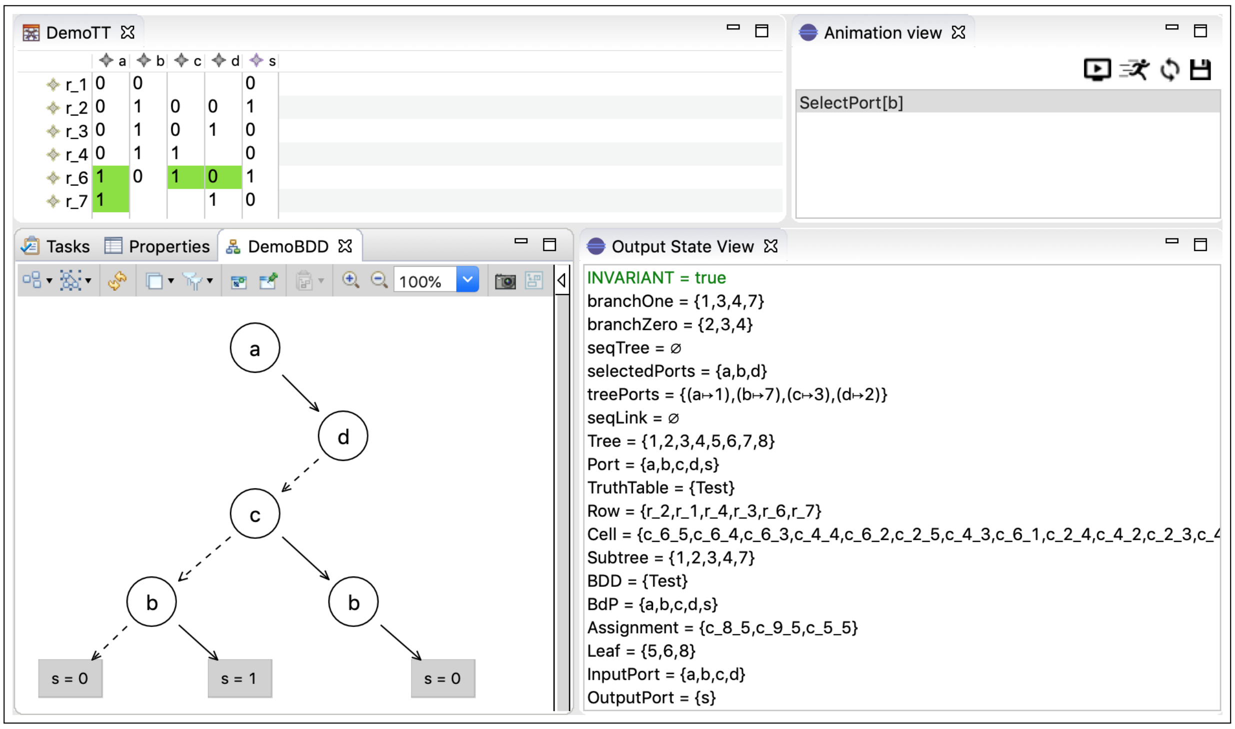
Our exhaustive model-checking validation technique was done on input models of reasonable sizes: until 5 input ports, 2 output ports, and 32 rows. We believe that the model-checking proofs done given these models are sufficient to have confidence about our rules because most of the provided models are 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.



	a	b	c	d	s
r_1	0	0			0
r_2	0	1	0	0	1
r_3	0	1	0	1	0
r_4	0	1	1		0
r_6	1	0	1	0	1
r_7	1			1	0

```
SelectPort[b]
```

```
INVARIANT = true
branchOne = {1,3,4,7}
branchZero = {2,3,4}
seqTree = ∅
selectedPorts = {a,b,d}
treePorts = {(a↦1),(b↦7),(c↦3),(d↦2)}
seqLink = ∅
Tree = {1,2,3,4,5,6,7,8}
Port = {a,b,c,d,s}
TruthTable = {Test}
Row = {r_2,r_1,r_4,r_3,r_6,r_7}
Cell = {c_6_5,c_6_4,c_6_3,c_4_4,c_6_2,c_2_5,c_4_3,c_6_1,c_2_4,c_4_2,c_2_3,c_
Subtree = {1,2,3,4,7}
BDD = {Test}
BdP = {a,b,c,d,s}
Assignment = {c_8_5,c_9_5,c_5_5}
Leaf = {5,6,8}
InputPort = {a,b,c,d}
OutputPort = {s}
```

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 ==
          LET cr BE cr = {cc,rr | rr ∈ Row ∧ cc= card(cells ⁻¹ [{rr}] ∩ selectedCells)} IN
            LET mx BE mx = max(dom(cr)) IN
              cr[{mx}]
            END
          END;

  portRow(rr) == (cellPort ⁻¹ ; cells) ▷ rr ;

  maxPort(pp,rr) == pp ∈ InputPort ∧ rr ⊆ Row ∧
      ¬ ( ∃ ss . (ss ∈ InputPort ∧ ss ≠ pp ∧ ss ∈ dom(portRow(rr))
         ∧ card(portRow(rr)[{ss}]) > card(portRow(rr)[{pp}]))) ;

  zeroCells(pp) == (cellPort ⁻¹ [{pp}] ∩ cells ⁻¹ [selectedRows]) ∩ Cell_value ⁻¹ [{FALSE}] ;

  oneCells(pp) == (cellPort ⁻¹ [{pp}] ∩ cells ⁻¹ [selectedRows]) ∩ Cell_value ⁻¹ [{TRUE}] ;

  selectedCells == dom(Cell_selected ▷ {TRUE}) ;

  outputCells(rr) == cells ⁻¹ [{rr}] ∩ cellPort ⁻¹ [OutputPort] ;

  inputCells(rr) == cells ⁻¹ [{rr}] ∩ 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* := ∅ , ∅ , ∅ ||
  treePorts, *seqTree*, *seqLink* := ∅ , ∅ , ∅ ||
  setLastTree(**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 first($seqLink$) = 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:

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

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

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

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

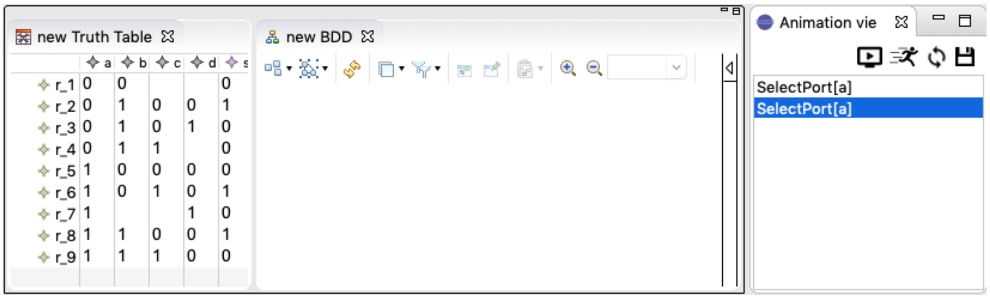
$$inputCells(rr) == cells^{-1}[\{rr\}] \cap cellPort^{-1}[InputPort]$$

- *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.



new Truth Table

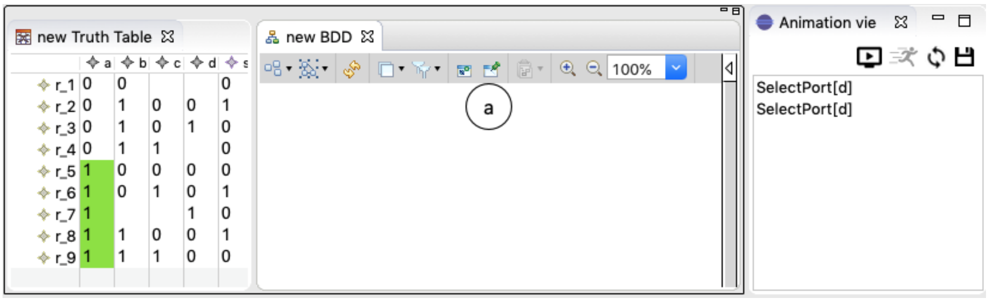
	a	b	c	d	s
r_1	0	0			0
r_2	0	1	0	0	1
r_3	0	1	0	1	0
r_4	0	1	1		0
r_5	1	0	0	0	0
r_6	1	0	1	0	1
r_7	1			1	0
r_8	1	1	0	0	1
r_9	1	1	1	0	0

SelectPort[a]
SelectPort[a]

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



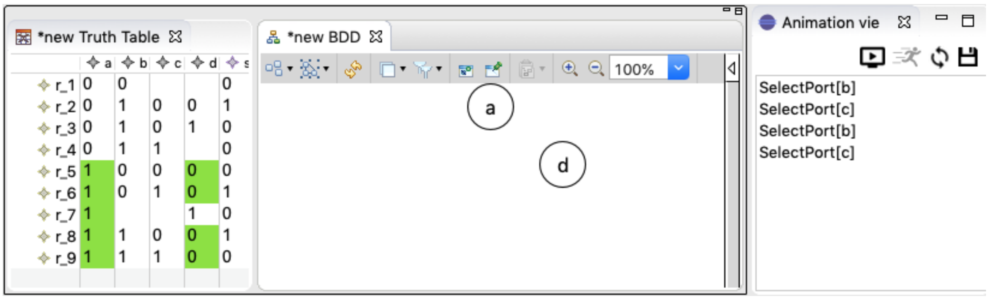
new Truth Table

	a	b	c	d	s
r_1	0	0			0
r_2	0	1	0	0	1
r_3	0	1	0	1	0
r_4	0	1	1		0
r_5	1	0	0	0	0
r_6	1	0	1	0	1
r_7	1			1	0
r_8	1	1	0	0	1
r_9	1	1	1	0	0

SelectPort[d]
SelectPort[d]

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.

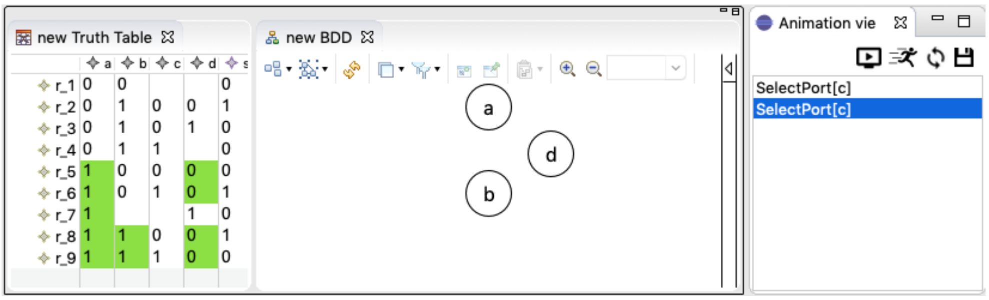


*new Truth Table

	a	b	c	d	s
r_1	0	0			0
r_2	0	1	0	0	1
r_3	0	1	0	1	0
r_4	0	1	1		0
r_5	1	0	0	0	0
r_6	1	0	1	0	1
r_7	1			1	0
r_8	1	1	0	0	1
r_9	1	1	1	0	0

SelectPort[b]
SelectPort[c]
SelectPort[b]
SelectPort[c]

Figure 5: Third execution



new Truth Table

	a	b	c	d	s
r_1	0	0			0
r_2	0	1	0	0	1
r_3	0	1	0	1	0
r_4	0	1	1		0
r_5	1	0	0	0	0
r_6	1	0	1	0	1
r_7	1			1	0
r_8	1	1	0	0	1
r_9	1	1	1	0	0

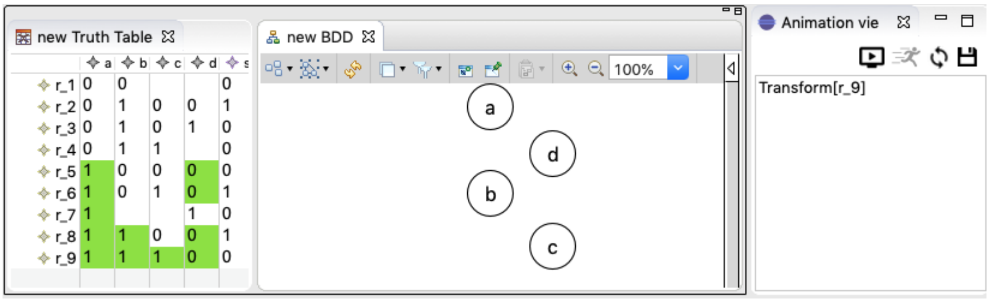
SelectPort[c]
SelectPort[c]

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.

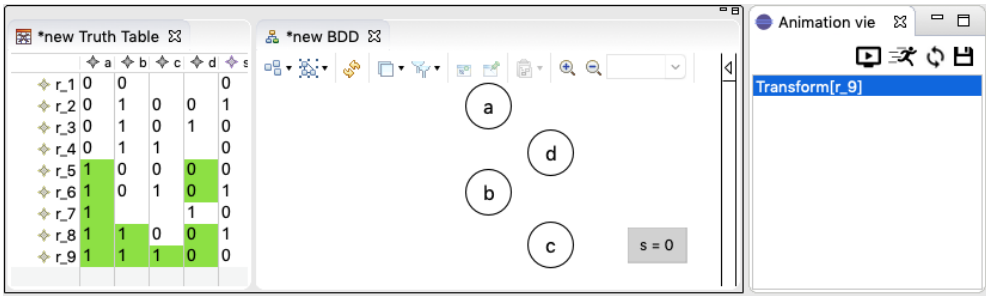
Figure 10 gives the resulting model after a row is entirely consumed and the corresponding path in the BDD is produced. Rule `continue` then updates the internal state of the B machine and makes possible the port selection process for the remaining rows. From the model of figure 10 only port `c` with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



	a	b	c	d	s
r_1	0	0			0
r_2	0	1	0	0	1
r_3	0	1	0	1	0
r_4	0	1	1		0
r_5	1	0	0	0	0
r_6	1	0	1	0	1
r_7	1			1	0
r_8	1	1	0	0	1
r_9	1	1	1	0	0

```
Transform[r_9]
```

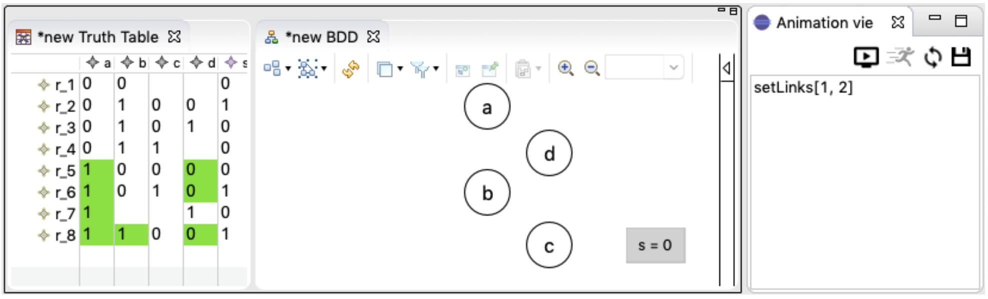
Figure 7: Fifth execution



	a	b	c	d	s
r_1	0	0			0
r_2	0	1	0	0	1
r_3	0	1	0	1	0
r_4	0	1	1		0
r_5	1	0	0	0	0
r_6	1	0	1	0	1
r_7	1			1	0
r_8	1	1	0	0	1
r_9	1	1	1	0	0

```
Transform[r_9]
```

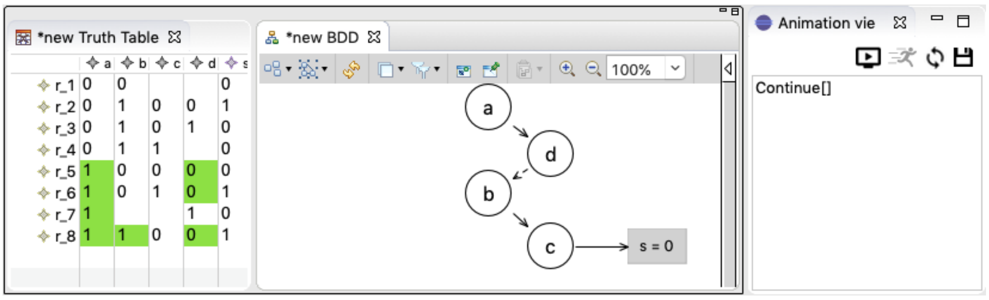
Figure 8: Sixth execution



	a	b	c	d	s
r_1	0	0			0
r_2	0	1	0	0	1
r_3	0	1	0	1	0
r_4	0	1	1		0
r_5	1	0	0	0	0
r_6	1	0	1	0	1
r_7	1			1	0
r_8	1	1	0	0	1

```
setLinks[1, 2]
```

Figure 9: Seventh execution



	a	b	c	d	s
r_1	0	0			0
r_2	0	1	0	0	1
r_3	0	1	0	1	0
r_4	0	1	1		0
r_5	1	0	0	0	0
r_6	1	0	1	0	1
r_7	1			1	0
r_8	1	1	0	0	1

```
Continue[]
```

Figure 10: A successive animation of setLink