YAMTL Solution to the TTC 2019 TT2BDD Case

Artur Boronat

Department of Informatics, University of Leicester, UK

aboronat@le.ac.uk

1 Introduction

In this paper, we present a solution for the TTC'19 TT2BDD case[1], where a set of boolean functions represented in a truth table with several input ports and several output ports is to be represented using binary decision diagrams (BDDs) [Bry86]. Our solution[2] is implemented using YAMTL [Bor18], an internal DSL of Xtend[3] that enables the use of model-to-model transformation within JVM programs, and produces a target model containing one reduced ordered BDD (ROBDD) [Bry86, Bry92] per boolean function[4], whose signature is given by the input ports and by one output port in the source truth table. In the rest of the paper, we discuss relevant aspects of our solution and we evaluate it according to the criteria proposed in the case.

2 Solution

The YAMTL solution is implemented with a model transformation that is executed in two stages, as illustrated in Fig. 1. In the first stage, the truth table is represented as a set of boolean expressions in disjunctive normal form (DNF), one per output port. In the second stage, the target model is built and a ROBDD is build from each of the boolean expressions.

A	B	C	D	S
0	0	-	-	0
0	1	0	0	1
0	1	0	0	1
0	1	0	1	0
0	1	1	-	0
1	0	0	0	0
1	0	1	0	1
1	-	-	1	0
1	1	0	0	1
1	1	1	0	0

$$\overline{A} \cdot B \cdot \overline{C} \cdot \overline{D} +$$
$$A \cdot \overline{B} \cdot C \cdot \overline{D} +$$
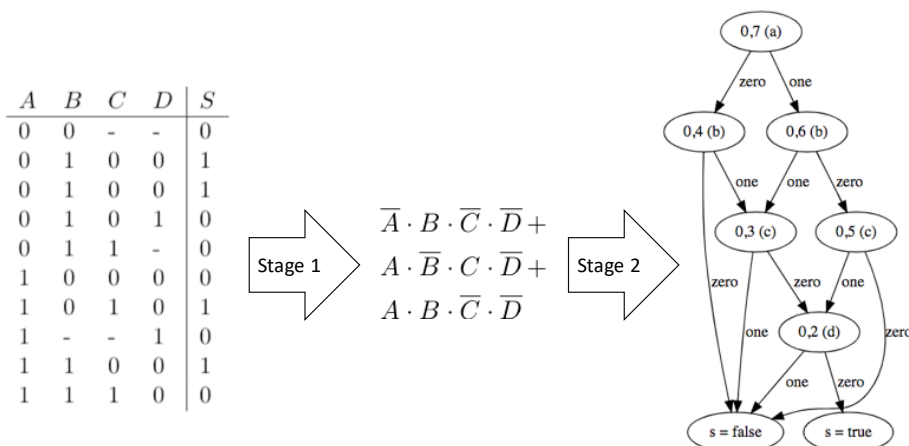$$A \cdot B \cdot \overline{C} \cdot \overline{D}$$

Figure 1: Outline of Solution

In the following subsections, the representation of boolean expressions used in the solution is presented. Then the two stages of the transformation are explained.

---

[1] https://www.transformation-tool-contest.eu/2019-tt2bdd.pdf

[2] Available at https://github.com/arturboronat/ttc2019-tt2bdd/tree/master/solutions/EMFSolutionYAMTL/

[3] In the rest of the paper, some familiarity with Xtend is assumed. In particular, no effort has been made at representing its dialect for handling collections as pseudocode.

[4] The upper bound of the reference `BDD::root` in the graph BDD metamodel (`BBDv2.ecore`) has been set to `*` to enable this.

2.1 Boolean Expressions and their Reduction

Boolean expressions are represented in disjunctive normal form (DNF), i.e., as a disjunction of conjunctive clauses. Each conjunctive clause is represented using a list of integers, where the index of a member identifies the input port and where the value 0 represents a negated port $\overline{A}$, and the value 1 represents a port. In the source truth table, a row that does not include any assignment for a port represents a set of rows, corresponding to rows containing all possible assignments for such port. This fact has been accounted for by including tautologies of the form $A + \overline{A}$ in conjunctive clauses, which are represented with the value 2, and the actual conjunctive clauses can be obtained using the $+$ distributive law. For example, the first row in the truth table of Fig. 1 is captured by the formula $\overline{A} \cdot \overline{B}$, which is represented as [0,0,2,2]. For each output port, a boolean expression is constructed as a disjunction of conjuntive clauses that are valid, i.e. whose output port value is 1. The disjunction is represented by using a list of conjunctive clauses. The example in Fig. 1 shows the boolean expression obtained for the given truth table, where the expression $\overline{A} \cdot \overline{B}$ is not included. `dnfMap` represents a global map of boolean expressions, indexed by output ports.

An important step in the construction of BDD graphs used in our solution consists in computing a successor after replacing a port with either 0 or 1. The reduction step, implemented in the listing below, involves replacing each port in a clause for its value with a subsequent cancellation of clauses that become invalid (line 3), when the port is negated $\overline{A}$ and the value is `true` or when the port A is not negated and the value is `false`. Cancellation is performed by removing the corresponding clause from the list (line 3). When a tautology of the form $A + \overline{A}$ is present in a clause, representing several clauses, then a conjunctive formula is cancelled (line 4). For example, if the value is `false`, then the formula with $\overline{A}$ would be preserved. When an expression `dnf` is reduced for all of its input ports, then each remaining element in the list `dnf` represents a list of assignments for input ports that satisfies a clause, thereby denoting that the expression `dnf` is valid. If no elements are left, then the boolean expression is not valid for the list of assignments used in reduction steps.

```
1 def reduce(List<List<Integer>> dnf, Port x, Integer value) {
2   val i = x.owner.ports.indexOf(x)
3   dnf.removeIf([r | r.get(i) != value])
4   dnf.filter[r | r.get(i)==2].forEach[r | r.set(i, value.complement)]
5   dnf
6 }
```

2.2 Stage 1: Obtaining Boolean Expressions in DNF

In the first stage, the rows of the source truth table are processed with a high priority rule `Row`, which builds a conjunctive clause from the values of the input ports in the row and, for each output port, completes the corresponding DNF expression in `dnfMap` if it is valid.

In YAMTL, a rule consists of an `in` pattern matching elements of the source model, which may contain filters, and of an `out` pattern, creating elements in the target model. Rule `Row` applies to rows in the truth table and creates an assignment in the target model. This rule does not have any side effect in the target model because it is declared as `transient` but it does initialize the `dnfMap` containing a DNF boolean expression per output port, as explained in the previous section. The YAMTL operator `fetch` at the start of the out element action (line 3) retrieves the value of the matched object `r`.

```
1 new Rule('Row').priority(1).transient // input cells are assumed to appear before output cells
2  .in('r', TT.row).build()
3  .out('a', BDD.assignment, [ val r = 'r'.fetch as Row
4    val portList = new ArrayList<ttc2019.metamodels.tt.Port>(r.owner.ports)
5    val oPortList = portList.filter[it instanceof ttc2019.metamodels.tt.OutputPort]
6      .map[it as ttc2019.metamodels.tt.OutputPort]
7    if (dnfMap.empty) // initialize list of boolean expressions, one per output port
8      for (oPort: oPortList) dnfMap.put(oPort, newArrayList)
9    val List<Integer> clause = newArrayList
10   for (c: r.cells) // each row yields a dnf formula
11     if (c.port != portList.head) { // a port is not set in the row
12       clause.add(2) // considered as a tautology: c.port or not(c.port)
13     } else {
14       portList.remove(c.port)
15       if (c.port instanceof ttc2019.metamodels.tt.InputPort)
16         clause.add(c.value.toInt) // false.toInt=0, true.toInt=1
17       else // when the dnf is valid, it is added to the corresponding expression
18         if (c.value) dnfMap.get(c.port).add(clause)
19     }
20  ]).build()/* for out element */.build()/* for rule */
```

2.3 Stage 2: Building ROBDDs

In the second stage, the BDD model with ports is created with the rules `TableToBDD`, `InputPort` and `OutputPort`, which are not transient. The class `AuxStruct` declares three auxiliary data structures: `nodeIndex`, defined for indexing nodes in ROBDDs; `nodeMap` for obtaining the index of a given ROBDD node; `treeMap` for caching nodes by the name of the input variable and by the indexes of the zero and one succesor nodes, `zeroIndex` and `oneIndex` resp., in order to avoid duplication.

The rule `TableToBDD` builds the root object of the model and creates input and output ports in the BDD model via the rules `InputPort` and `OutputPort` respectively. The YAMTL operator `fetch` at the start of the out element action (line 4) retrieves the value of the matched truth table `tt` and of the created out BDD object `bdd` (and similarly for the other rules). The YAMTL operator fetch, in lines 5-6, resolves a list of references to elements in the source model by returning a list of references to elements in the target model, as they are transformed by other rules. In particular, YAMTL will automatically find that these references are resolved with the rule `InputPort` in line 5 and by the rule `OutputPort` in line 6. The rule `TableToBDD` also uses the helper `build`, which returns the root tree nodes of each ROBDD corresponding to each output port in the truth table.

```
1 new Rule('TableToBDD')
2   .in('tt', TT.truthTable).build()
3   .out('bdd', BDD.BDD, [
4     val tt = 'tt'.fetch as TruthTable; val bdd = 'bdd'.fetch as BDD
5     bdd.ports += tt.ports.fetch('bdd_p','InputPort') as List<InputPort>
6     bdd.ports += tt.ports.fetch('bdd_p','OutputPort') as List<OutputPort>
7     bdd.root += dnfMap.build(bdd)
8   ]).build()
9 .build(),
10
11 new Rule('InputPort')
12   .in('tt_p', TT.inputPort).build()
13   .out('bdd_p', BDD.inputPort, [
14     val tt_p = 'tt_p'.fetch as tt.InputPort; val bdd_p = 'bdd_p'.fetch as InputPort
15     bdd_p.name = tt_p.name
16   ]).build()
17 .build(),
18
19 new Rule('OutputPort')
20   .in('tt_p', TT.outputPort).build()
21   .out('bdd_p', BDD.outputPort, [
22     val tt_p = 'tt_p'.fetch as tt.OutputPort; val bdd_p = 'bdd_p'.fetch as OutputPort
23     bdd_p.name = tt_p.name
24   ]).build()
25 .build()
```

For each boolean expression in `dnfMap`, the helper `build` initializes the auxiliary data structures in an `AuxStruct` instance and calls the helper `build` for that particular expression (`entry.value`) returning the root of the generated ROBDD, which is then added in the `rootTreeList` (line 6 below).

```
1 def build(Map<tt.OutputPort,List<List<Integer>>> dnfMap, BDD bdd) {
2   var List<Tree> rootTreeList = newArrayList
3   val iPortList = bdd.ports.filter[it instanceof InputPort].map[it as InputPort].toList
4   for (entry: dnfMap.entrySet) {
5     val oPort = bdd.ports.findFirst[it.name==entry.key.name] as OutputPort
6     rootTreeList.add(entry.value.build(iPortList, oPort, new AuxStruct))
7   }
8   rootTreeList
9 }
```

The helper `build` for a particular boolean expression `dnf`, in the listing below, takes the next input port from `iPortList` to be processed (line 2), and builds the ROBDD for the assignments `0` (in line 5) and `1` (in line 6) using the helper `buildTree`, which returns the root of the corresponding ROBDD and indexes the nodes of the ROBDD. When the indexes of the root nodes for both branches coincide, then the two branches are the same and no new node needs to be added (line 11-12). Otherwise, a new node is created and cached if it has not been indexed by input port name and indexes of successors (lines 16-19). When the node to be created has been indexed already (the `triple` in line 15 is not null), then the subtree has already been created and it does not need to be duplicated. The helper `createSubtree` instantiates the class `Subtree` and initializes its variable and children, indexing it in `treeMap`.

```
1 def build(List<List<Integer>> dnf, List<InputPort> iPortList, OutputPort oPort, AuxStruct aux) {
2   val iPort = iPortList.head
3   val portTail = iPortList.tail.toList
4   // create successors
5   var tree0 = dnf.clone().buildTree(iPort, portTail, oPort, 0, aux)
6   var tree1 = dnf.clone().buildTree(iPort, portTail, oPort, 1, aux)
7   // create tree by adding root
8   var Tree tree
9   val zeroIndex = aux.nodeIndex.get(tree0)
10  val oneIndex = aux.nodeIndex.get(tree1)
11  if (zeroIndex == oneIndex) { // avoid redundant tests
12    tree = tree0
13  } else { // create tree if not yet indexed by port name, zero index and one index
14    val triple = new ImmutableTriple(iPort.name, zeroIndex, oneIndex)
15    tree = aux.treeMap.get(triple)
16    if (tree === null) {
17      tree = iPort.createSubtree(tree0, tree1, aux) // creates a subtree with root x and two children
18      aux.treeMap.put(triple,tree)
19    }
20  }
21  tree
22 }
```

The helper `buildTree`, in the listing below, reduces the boolean expression `dnf` by substituting `value` for the input port `iPort` (line 3), obtaining `reducedDNF`, and continues processing the next input port by calling `build` for `reducedDNF` (line 5). If the list of remaining input ports `portTail` is empty, the helper returns a `zero` leaf (line 8) when there are no assignments for the expression `dnf`, i.e. the list is empty meaning that the DNF is not satisfiable, and a `one` leaf (line 10) when there are assignments, i.e. the list `dnf` contains the assignments for the input ports for which the expression is valid. The `createLeaf` creates a `Leaf` instance, initializing its value and indexing the node.

```
1 def Tree buildTree(List<List<Integer>> dnf, Port iPort, List<InputPort> portTail, OutputPort oPort, Integer
      value, AuxStruct aux) {
2   var Tree tree
3   val reducedDNF = dnf.reduce(iPort, value) // applies substitution value/x and cancels invalid dnfs
4   if (!portTail.empty) { // there are still more variables to be reduced
5     tree = reducedDNF.build(portTail, oPort, aux)
6   } else { // no variables left
7     if (reducedDNF.empty) // no DNFs left in the expression: the result is 0
8       tree = oPort.createLeaf(0, aux) // helper that creates leaf node for oPort and indexes it at 0
9     else // the result is 1
10      tree = oPort.createLeaf(1, aux) // helper that creates leaf node for oPort and indexes it at 1
11  }
12  tree
13 }
```

3   Evaluation and Conclusions

Below we discuss the criteria proposed to evaluate the solution, which we conclude by highlighting the aspects of YAMTL that have been exploited in this solution.

Input Model		ATL (tree)				YAMTL			
input	output	size	init	load	run	size	init	load	run
4	1	32	256.9	118.8	457.6	16	247.0	110.5	59.8
4	2	70	253.0	124.5	594.5	30	245.4	126.1	66.9
8	2	1034	249.9	220.0	53.7″	168	245.1	252.8	136.2
10	2	4108	252.1	375.7	1330.8″	503	256.7	404.9	222.0
12	2					1489	242.5	914.1	397.5
14	4					9193	248.9	2.3″	1.1″
15	5					21592	244.5	4.3″	2.5″

Table 1: Model element cardinalities and execution times in ms. (unless stated ″ for s.)

3.1 Correctness

As explained above, each output port in the final BDD model corresponds to the output of a boolean function, which is represented as a separate ROBDD. The correctness of our solution has been checked with a generalization of the proposed graph validator, which runs the validator for each of the ROBDDs that have been generated and that are available under `BDD::root` (with upper bound `*`).

3.2 Completeness

The solution can be executed for all of the sample input truth table models provided, including up to 15 input ports and up to 5 output ports.

3.3 Performance

For input models, Table 1 shows the number of *input* ports and *output* ports. The table also shows the size of the resulting BDD model (object cardinality) and execution times for each of the phases (*init*ialization, *load*, *run*) both for the reference solution (tree metamodel) and for the YAMTL solution.

For the asymptotic analysis of our solution, we consider n input ports and m output ports in the source truth table. The algorithm in our solution is split in two parts. First, the truth table is linearly processed in order to build boolean expressions, which involves up to n^2 rows consisting of n input cells, that is $\mathcal{O}(n^3)$. Second, each boolean expression consisting of n input ports is simplified, once for each output port, that is $n \times m$. Simplification involves reducing each port in all clauses of a DNF expression. In case of a tautology, this involves making substitutions in n^2 clauses! Therefore, the worst time complexity of the second part is $\mathcal{O}(m \times n^3)$, and the worst time for the overall solution is $\mathcal{O}(n^3) + \mathcal{O}(m \times n^3)$. However, as shown in Fig. 1, tautologies are a rare case and we can assume that the second part will tend towards the best time complexity, which is $\mathcal{O}(m \times n)$, during the reduction of each boolean expression. Hence, average performance is increasingly dominated by the processing of the truth table in the first part as the number of input ports grow and, in practice, our solution tends to be linear in terms of the number of cells, corresponding to input ports, in the source truth table, that is $\mathcal{O}(n^3)$.

3.4 Optimality

Our BDD models correspond to lists ROBBDs, one per output port, where all paths through the graph respect the linear order of input ports, as defined in the input table, and all nodes are unique (no two distinct nodes have the same variable name and zero and one successors) and contain no redundant tests (no variable node has identical zero and one successors). The result is a compact representation of a boolean function, as can be observed in the size column of the solutions in Table 1.

3.5 Conclusion and Research Direction

The solution presented is correct and complete w.r.t. the evaluation criteria provided in the case. When comparing it with the reference ATL transformation (on the tree metamodel), in Table 1, the transformation shows good performance and produces smaller output models. Moreover, the solution presented showcases several features of YAMTL: interoperability with Java data structures via Xtend; application of rules in stages using priorities; declarative rules with side effects, one of which is transient.

References

[Bor18] Artur Boronat. Expressive and efficient model transformation with an internal dsl of xtend. In *Proceedings of the 21th ACM/IEEE International Conference on MoDELS*, pages 78–88. ACM, 2018.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.