T-79.186 Reactive Systems

Spring 2003, Lecture 7

Keijo Heljanko

February 26, 2003

The top-level LTL to Büchi automaton translation algorithm just does some initialization, and then calls the "expand(node)" subroutine.

Algorithm 1 The top-level LTL to Büchi translation algorithm

global nodes: Set of Node; // Use e.g., a hash table
procedure translate(f: Formula)
  local node: Node;
  nodes := $\emptyset$; // Initialize the result to empty set
  node := NewNode(); // Allocate memory for a new node
  node.ID := GetID(); // Allocate a new node ID
  node.Incoming := $\{init\}$; // Incoming can be implemented as a list
  node.New = $\{f\}$; // Use e.g., bit-arrays of size $sub(f)$ for these sets
  node.Old = $\emptyset$;
  node.Next = $\emptyset$;
  expand(node); // Call the recursive expand procedure
  return nodes;
end procedure

Algorithm 2 LTL to Büchi translation main loop

```
procedure expand(node: Node)
    local node, node1, node2: Node;
    local f: Formula;
    if node.New = ∅ then
        if ∃ node1 ∈ nodes with node1.Old = node.Old ∧ node1.Next = node.Next then
            node1.Incoming := node1.Incoming ∪ node.Incoming; // redirect arcs to "node1"
            return; // Discard "node" by not storing it to "nodes"
        else
            nodes := nodes ∪{node }; // "node" is ready, add it to the automaton
            node2 := NewNode(); // Create "node2" to prove formulas in "node.Next"
            node2.ID := GetID();
            node2.Incoming := { node.ID };
            node2.New = { node.Next };
            node2.Old = ∅;
            node2.Next = ∅;
            expand(node2);
            return;
```

```
    else // node.New ≠ ∅ holds
        pick f from node.New; // Any formula "f" in "node.New" will do
        node.New := node.New \{ f }; // Remove "f" from proof objectives
        switch begin(FormulaType(f))
            case atomic proposition, negated atomic proposition, true, false:
                expand_simple(node,f);
            return;
            case conjunction:
                expand_conjunction(node,f);
            return;
            case disjunction, until, release:
                expand_disjunction(node,f);
            return;
        switch end
    // Not reached
    return;
end procedure
```

Algorithm 3 Expanding simple formulas

```
procedure expand_simple(node: Node, f: Formula)
    if f = false or Neg(f) ∈ node.Old then
        return; // "node" contains a contradiction (false / both p and ¬p), discard it
    else
        node.Old := node.Old ∪ { f }; // Recall that this node proves "f"
        expand(node); // Handle also other formulas in "node.New"
    return;
end procedure
```

Algorithm 4 Expanding conjunction

```
procedure expand_conjunction(node: Node, f: Formula)
    local f1, f2: Formula;
    f1 := left(f); // Obtain subformula "f1" from left side of f1 ∧ f2
    f2 := right(f); // Obtain subformula "f2" from right side of f1 ∧ f2
    node.New := node.New ∪ ({ f1, f2 } \ node.Old); // Prove both "f1" and "f2"
    node.Old := node.Old ∪ { f }; // Recall that this node proves "f"
    expand(node); // Handle also other formulas in "node.New"
    return;
end procedure
```

Algorithm 5 Expanding disjunction

procedure expand_disjunction(node: Node, f: Formula)

local f1, f2: Formula;

local node1, node2: node;

// This one handles all the cases: $f_1 \vee f_2$, $f_1 \, U \, f_2$, $f_1 \, R \, f_2$

// Replace "node" with two nodes "node1" and "node2" (The blow-up happens here!)

// Do the proof using strategy (b)

node1 := NewNode(); // Create "node1" to prove formulas using strategy (b)

node1.ID := GetID();

node1.Incoming := node.Incoming;

node1.New = node.New ∪ ($\mathit{New1}(f) \setminus$ node.Old); // Prove things in $\mathit{New1}(f)$

node1.Old := node.Old ∪ { f }; // Recall that "node1" node proves "f"

node1.Next = node.Next ∪ $\mathit{Next1}(f)$; // On the next time, prove things in $\mathit{Next1}(f)$

```
    // Do the proof using strategy (a)

    node2 := NewNode(); // Create "node2" to prove formulas using strategy (a)
    node2.ID := GetID();
    node2.Incoming := node.Incoming;
    node2.New = node.New ∪ (New2(f) \ node.Old); // Prove things in New2(f)
    node2.Old := node.Old ∪ { f }; // Recall that "node2" node proves "f"
    node2.Next = node.Next; // In case (a) Next2(f) is always empty

    expand(node1); // "node1" does the proof using strategy (b)
    expand(node2); // "node2" does the proof using strategy (a)
    return; // discard "node" by not storing it to "nodes"
end procedure
```