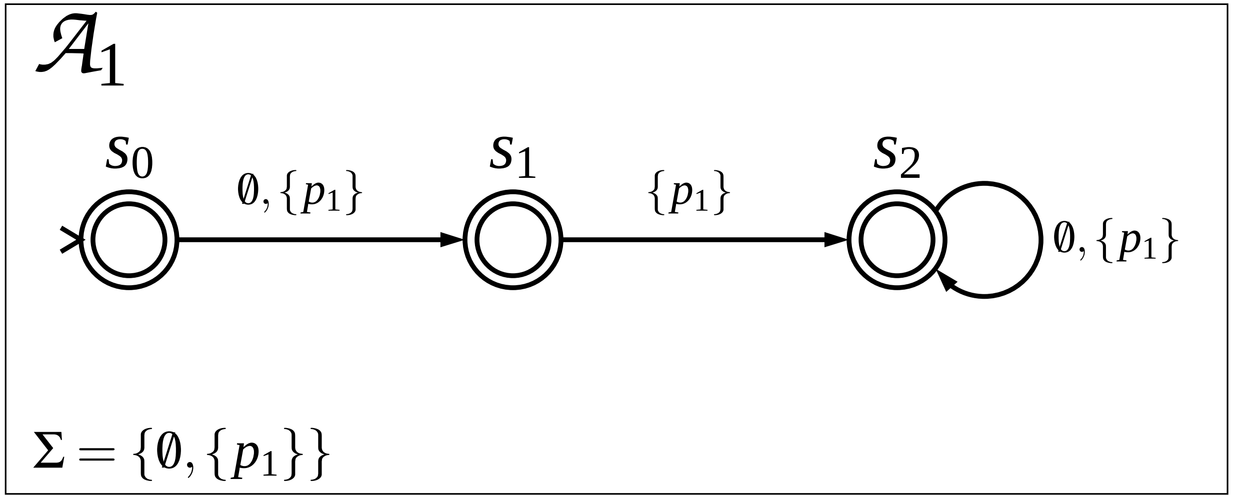
Reactive Systems: Büchi Automata and LTL

Timo Latvala

March 17, 2004

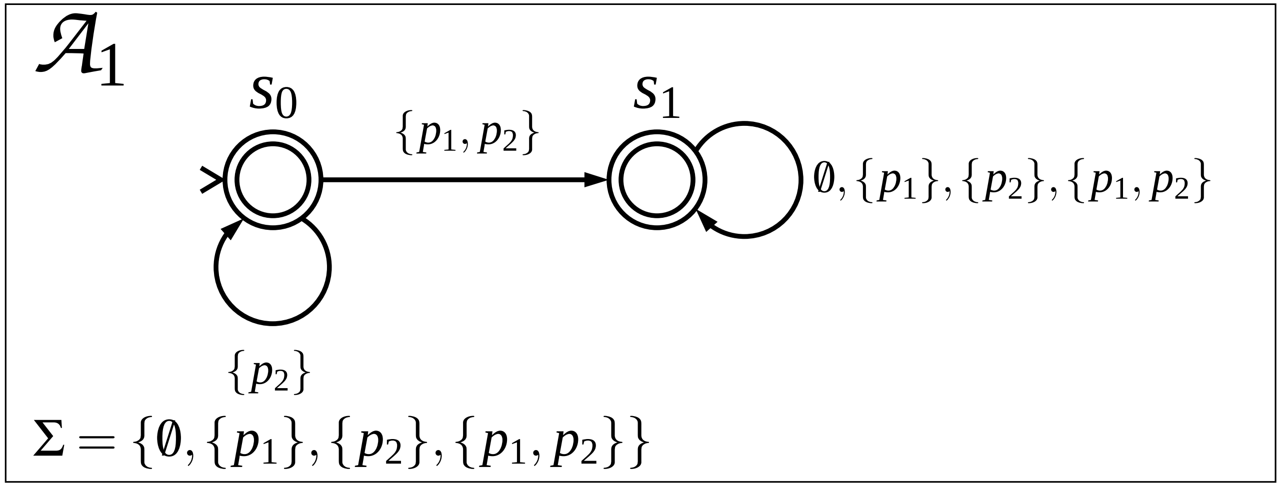
Example: $X\ p_1$ as Büchi Automaton

It is easy to see that the Büchi automaton $\mathcal{A}_1$ below will accept exactly the set of infinite words, which are models of the LTL formula $X\ p_1$.



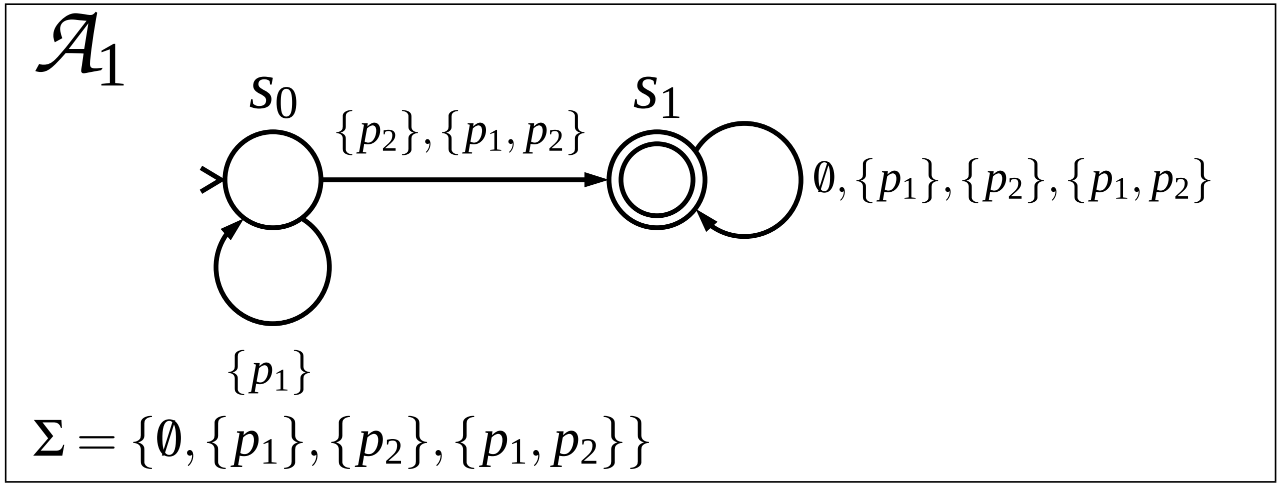
Example: $p_1 \, R \, p_2$ as Büchi Automaton

It is easy to see that the Büchi automaton $\mathcal{A}_1$ below will accept exactly the set of infinite words, which are models of the LTL formula $p_1 \, R \, p_2$.



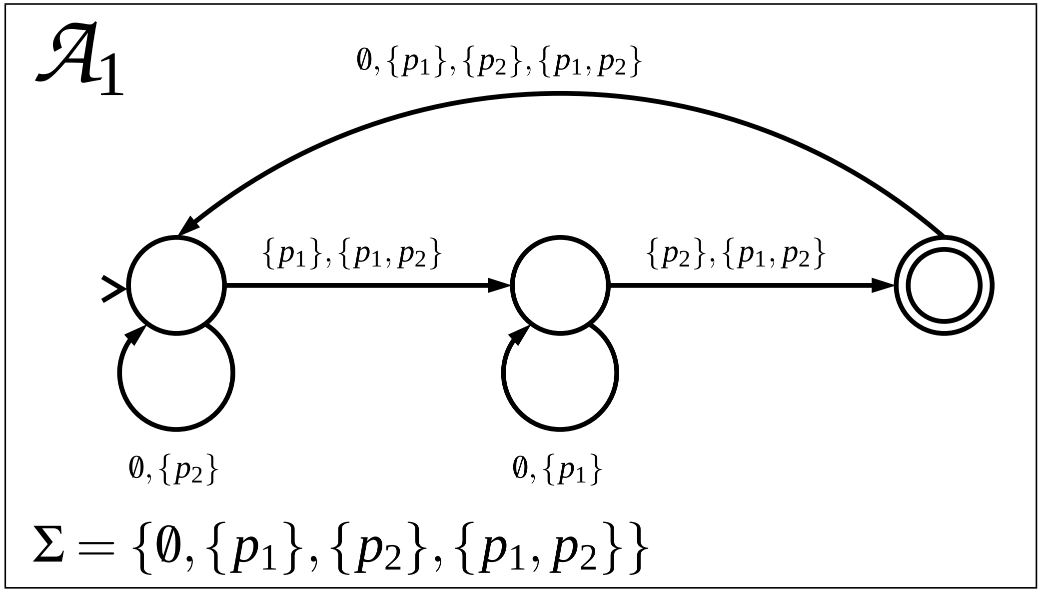
Example: $p_1 U p_2$ as Büchi Automaton

It is easy to see that the Büchi automaton $\mathcal{A}_1$ below will accept exactly the set of infinite words, which are models of the LTL formula $p_1 U p_2$.



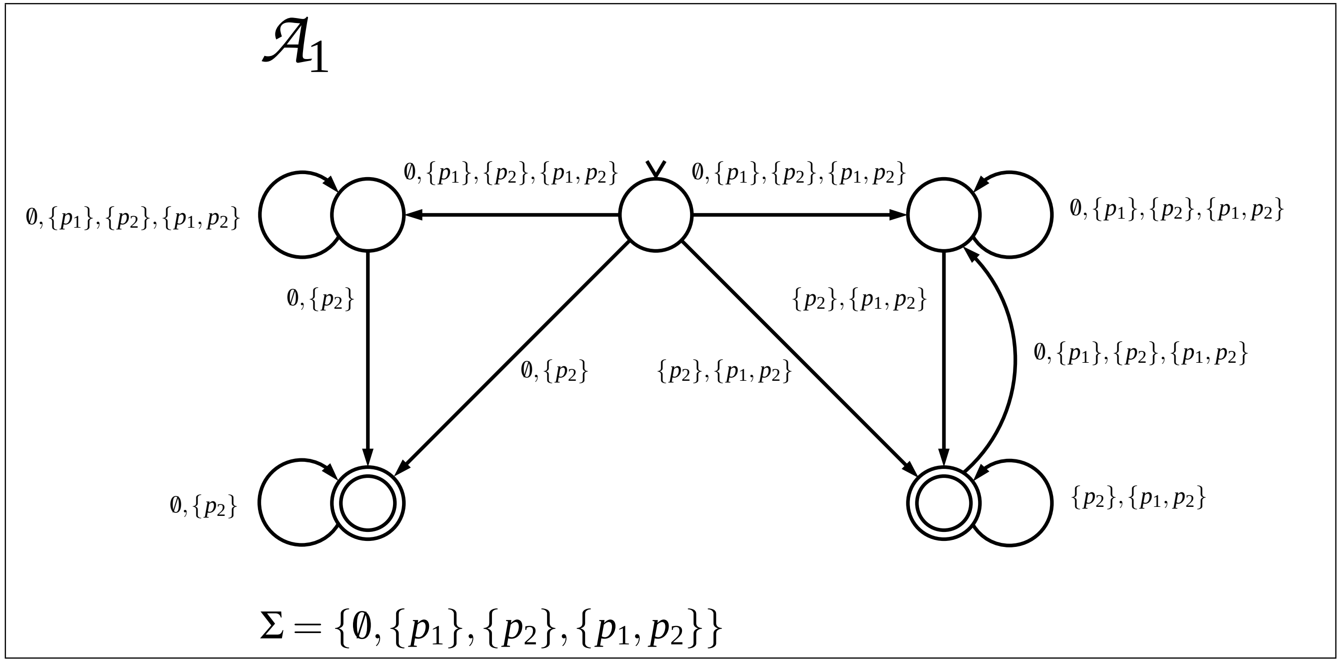
Example: $(\Box\Diamond p_1) \wedge (\Box\Diamond p_2)$ as Büchi Automaton

It is easy to see that the Büchi automaton $\mathcal{A}_1$ below will accept exactly the set of infinite words, which are models of the LTL formula $(\Box\Diamond p_1) \wedge (\Box\Diamond p_2)$.



Example: $(\Box\Diamond p_1) \Rightarrow (\Box\Diamond p_2)$ as Büchi Automaton

It is easy to see that the Büchi automaton $\mathcal{A}_1$ below will accept exactly the set of infinite words, which are models of the LTL formula $(\Box\Diamond p_1) \Rightarrow (\Box\Diamond p_2)$.



Translating LTL into Büchi Automata

There are several algorithms for translating LTL formulas into Büchi automata. In this course we will go through a variant due to Gerth, Peled, Vardi, and Wolper.

Given an LTL formula f, it will generate a Büchi automaton $\mathcal{A}_f$ of with at most $2^{O(|f|)}$ states.

The automaton $\mathcal{A}_f$ will accept the language $\{w \in \Sigma^\omega \mid w \models f\}$, where $\Sigma = 2^{AP}$.

Recall that if we want to model check an LTL property h, we should actually create an automaton for $f = \neg h$.

Before we proceed any further, we want to put the formula f into negation normal form (also called positive normal form), where all negations appear appear only in front of atomic propositions.

This can be done with previously presented DeMorgan rules for temporal logic operators. Note that putting the formula into positive normal form does not involve a blow-up.

We will keep f as the name of the formula after this procedure.

We will now define the set of formulas $sub(f)$ to be the smallest set of LTL formulas satisfying all of the following conditions:

- Boolean constants $\mathbf{true}, \mathbf{false}$, and the top-level formula f belong to $sub(f)$,
- if $f_1 \vee f_2 \in sub(f)$, then $\{f_1, f_2\} \subseteq sub(f)$
- if $f_1 \wedge f_2 \in sub(f)$, then $\{f_1, f_2\} \subseteq sub(f)$
- if $X\, f_1 \in sub(f)$, then $\{f_1\} \subseteq sub(f)$
- if $f_1 U f_2 \in sub(f)$, then $\{f_1, f_2\} \subseteq sub(f)$
- if $f_1 R f_2 \in sub(f)$, then $\{f_1, f_2\} \subseteq sub(f)$

It is easy to show that $|sub(f)| = O(|f|)$.

To ease implementation, the formulas of $sub(f)$ can be numbered, and thus any subset of $sub(f)$ can be represented with a bit-array of length $|sub(f)|$. In fact, there are at most $2^{O(|f|)}$ such subsets.

The basic idea of the translation is based on the following properties of the semantics of LTL, where we can choose which way to prove a particular property:

- To prove that $w \models f_1 \vee f_2$ it suffices to either prove that

  a) $w \models f_1$, or

  b) $w \models f_2$.

- To prove that $w \models f_1 U f_2$ it suffices to either prove that

  a) $w \models f_2$, or

  b) $w \models f_1$ and $w \models X(f_1 U f_2)$.

- To prove that $w \models f_1 R f_2$ it suffices to either prove that

  a) $w \models f_1$ and $w \models f_2$, or

  b) $w \models f_2$ and $w \models X(f_1 R f_2)$.

The only restriction being, that when proving $f_1 U f_2$ the case b) can only be used infinitely often iff the case a) is also used infinitely often (we will use Büchi acceptance sets to handle that).

The algorithm will manipulate a data structure called *node* during its run.

A node is a structure with the following fields:

- ID: A unique identifier of a node (a number),
- $Incoming$: A list of node IDs,
- $Old \subseteq sub(f)$,
- $New \subseteq sub(f)$, and
- $Next \subseteq sub(f)$.

The nodes will form a graph, where the arcs of the graph are stored in the *Incoming* list of the end node of the arc for easier manipulation.

The initial node is marked by having a special node ID called *init* in its *Incoming* list.

All nodes are stored in a set (use e.g., a hash table for implementation) called *nodes*.

To implement the algorithm, the following functions are defined.

- $\mathit{Neg}(\mathbf{true}) = \mathbf{false}$,
- $\mathit{Neg}(\mathbf{false}) = \mathbf{true}$,
- $\mathit{Neg}(p) = \neg p$ for $p \in AP$, and
- $\mathit{Neg}(\neg p) = p$ for $p \in AP$.

The functions $New1(f)$, $Next1(f)$, and $New2(f)$ are tabulated below. They match the recursive definitions for disjunction, until, release, and next-time:

f	$New1(f)$	$Next1(f)$	$New2(f)$
$f_1 \vee f_2$	$\{f_2\}$	$\emptyset$	$\{f_1\}$
$f_1 U f_2$	$\{f_1\}$	$\{f_1 U f_2\}$	$\{f_2\}$
$f_1 R f_2$	$\{f_2\}$	$\{f_1 R f_2\}$	$\{f_1, f_2\}$
$X f_1$	$\emptyset$	$\{f_1\}$	$\emptyset$

We are now ready to present the translation algorithm.

The top-level algorithm just does some initialization, and then calls “expand(node)”.

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 = $\emptyset$ **then**
        if $\exists$ node1 $\in$ nodes with node1.Old = node.Old $\wedge$ node1.Next = node.Next **then**
            node1.Incoming := node1.Incoming $\cup$ node.Incoming; // redirect arcs to “node1”
            return; // Discard “node” by not storing it to “nodes”
        else
            nodes := nodes $\cup \{$ 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 = $\emptyset$;
            node2.Next = $\emptyset$;
            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;
            case next:
                expand_next(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 $Neg(f) \in$ node.Old **then**
    return; // "node" contains a contradiction (**false** / both p and $\neg p$), discard it
  else
    node.Old := node.Old $\cup$ { f }; // Recall that this node proves "f"
    expand(node); // Handle the rest of the 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 $f_1 \wedge f_2$
f2 := right(f); // Obtain subformula “f2” from right side of $f_1 \wedge f_2$
node.New := node.New $\cup$ ($\{$ f1, f2 $\}$ $\setminus$ node.Old); // Prove both “f1” and “f2”
node.Old := node.Old $\cup$ $\{$ f $\}$; // Recall that this node proves “f”
expand(node); // Handle the rest of the 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 $\cup$ (*New1*$(f) \setminus$ node.Old); // Prove things in *New1*(f)
node1.Old := node.Old $\cup \{$ f $\}$; // Recall that “node1” node proves “f”
node1.Next = node.Next $\cup$ *Next1*(f); // On the next time, prove things in *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
```

Algorithm 6 *Expanding next*

procedure expand_next(node: Node, f: Formula)
  local f1: Formula;
  f1 := left(f); // Obtain subformula “f1” from $X\ f_1$

  // This one handles the case $X\ f_1$

  node.Old := node.Old ∪ { f }; // Recall that “node” node proves “f”
  node.Next = node.Next ∪ $\mathit{Next1}(f)$; // On the next time, prove things in $\mathit{Next1}(f)$

  expand(node); // Handle the rest of the formulas in “node.New”
  return;
end procedure