T-79.186 Reactive Systems

Büchi Automata and LTL

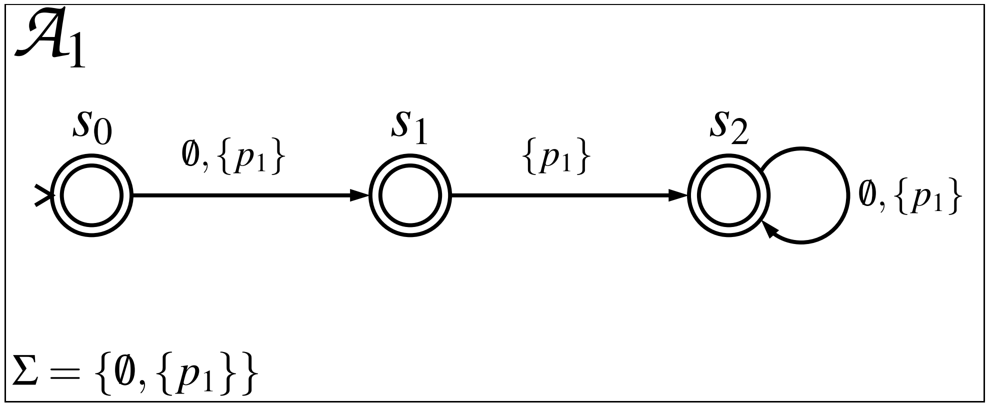
Spring 2005, Lecture 7

Keijo Heljanko

Keijo.Heljanko@tkk.fi

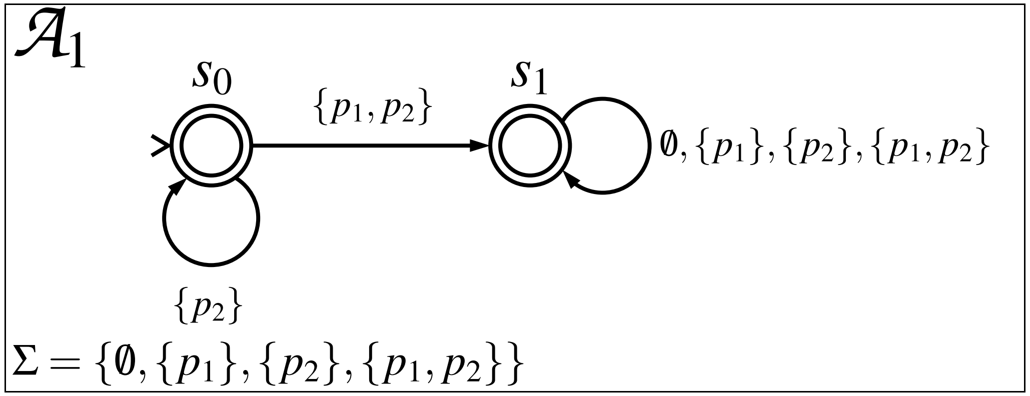
Example: $X\, p_1$ as Büchi Automaton

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 $X\, p_1$.



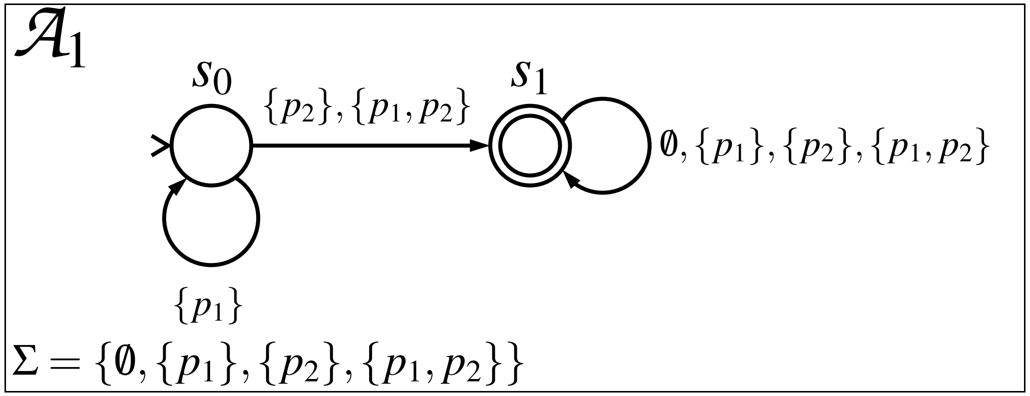
Example: $p_1 \, R \, p_2$ as BA

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 $p_1 \, R \, p_2$.


$\mathcal{A}_1$
s_0
s_1
$\{p_1, p_2\}$
$\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}$
$\{p_2\}$
$\Sigma = \{\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}\}$

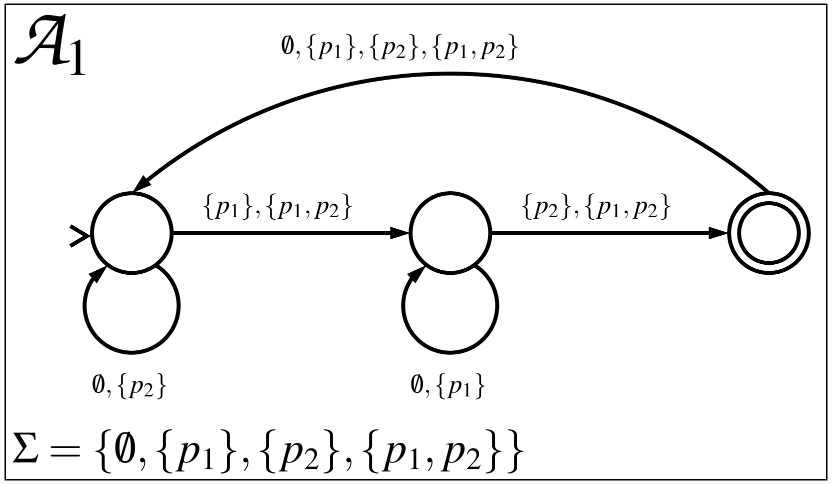
Example: $p_1\, U\, p_2$ as BA

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 $p_1\, U\, p_2$.


$\mathcal{A}_1$
s_0
s_1
$\{p_2\}, \{p_1, p_2\}$
$\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}$
$\{p_1\}$
$\Sigma = \{\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}\}$

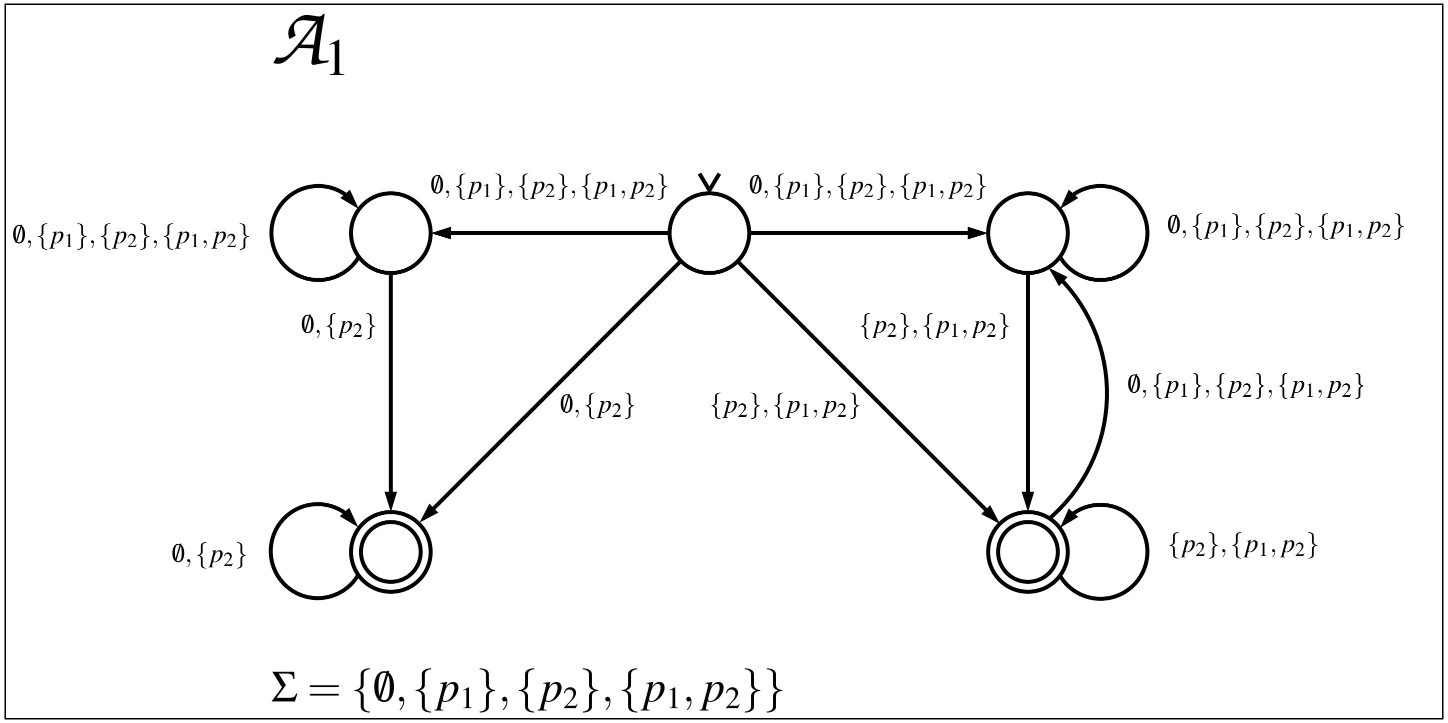
Example: $(\Box\Diamond p_1) \land (\Box\Diamond p_2)$ as BA

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 $(\Box\Diamond p_1) \land (\Box\Diamond p_2)$.



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

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 $(\Box\Diamond p_1) \Rightarrow (\Box\Diamond p_2)$.



$\mathcal{A}_1$
$\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}$
$\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}$
$\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}$
$\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}$
$\emptyset, \{p_2\}$
$\{p_2\}, \{p_1, p_2\}$
$\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}$
$\emptyset, \{p_2\}$
$\{p_2\}, \{p_1, p_2\}$
$\emptyset, \{p_2\}$
$\{p_2\}, \{p_1, p_2\}$
$\Sigma = \{\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}\}$

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 $f = \neg h$.
- First the formula f is transformed 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 without a blow-up.

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

- 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.

Proof Rules

The basic idea of the translation is based on the following properties of the semantics of LTL:

- 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),
- $\mathit{Incoming}$: A list of node IDs,
- $\mathit{Old} \subseteq \mathit{sub}(f)$,
- $\mathit{New} \subseteq \mathit{sub}(f)$, and
- $\mathit{Next} \subseteq \mathit{sub}(f)$.

- 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; f: Formula;
    if node.New = $\emptyset$ then
        if $\exists$ node1 $\in$ nodes with node1.Old = node.Old $\wedge$ node1.Next = node.Next the
            node1.Incoming := node1.Incoming $\cup$ node.Incoming; // redirect arcs
            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 $\neq \emptyset$ 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); break;
                case conjunction: expand_conjunction(node,f); break;
                case disjunction, until, release: expand_disjunction(node,f); break;
                case next: expand_next(node,f); break;
            switch end
        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 (false or both $p$ and $\neg p$)
    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 ∪ ({ f1, f2 } \ node.Old); // Prove both
    node.Old := node.Old ∪ { 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” (Blow-up is 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 ∪ ($New1(f) \setminus$ node.Old); // Prove things in $New1(f)$
    node1.Old := node.Old ∪ { f }; // Recall that “node1” node proves “f”
    node1.Next = node.Next ∪ $Next1(f)$; // 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 ∪ ($New2(f) \setminus$ 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
```

- The algorithm creates a graph stored in the set "$nodes$". The nodes of this graph are labeled with formulas. Actually, from now on we are only interested in the formulas stored in the set "Old".
- It is now easy to obtain a Büchi automaton from this graph. (Using a slightly different Büchi automata definition than what has been used in this course so far.)
- First of all a special initial state "$init$" is created. This state is the only state in S^0.
- All nodes "$p \in$ nodes" together with the initial state "$init$" are the states S of the Büchi automaton.

The labeling of the arcs of the Büchi automaton can be derived from the formula labeling of states.
Namely, a state is compatible with a set of valuations as described below.

A valuation $v \in 2^{AP}$ is compatible with the label of a node s iff:

- $\forall p \in AP$: if $p \in$ s.Old then $p \in v$, and
- $\forall p \in AP$: if $\neg p \in$ s.Old then $p \notin v$.

There is an arc from a state $s \in S$ to a state $r \in S$ with a label $v \in 2^{AP}$ iff

- v is compatible with the valuation of r, and
- $s \in r$.Incoming.

The Büchi automaton class used is called *generalized Büchi automata*. In this class the acceptance component consist of several acceptance sets. The basic idea is that an accepting run should visit some accepting state from each acceptance set infinitely often.

More formally the acceptance component is $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$, where each $F_i \subseteq S$.

Now a generalized Büchi automaton $\mathcal{A}$ accepts a run r iff for all $F_i \in \mathcal{F}$: $\mathit{inf}(r) \cap F_i \neq \emptyset$.

Note that in the special case $\mathcal{F} = \emptyset$ all the infinite runs of the generalized Büchi automaton are accepting.

We will to rule out the use of case b) infinitely many times without also using case a) infinitely many times when proving the until formula $f_1 U f_2 \in sub(f)$. This is done by using one acceptance set for each until formula.

Assume, that all the until subformulas (subformulas of the form $g_1 \, U \, g_2$) are $f_1, f_2, \ldots, f_n$. Then the acceptance component $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$, where:
For each $1 \leq i \leq n$ the state s belongs to F_i iff

- $right(f_i) \in s$.Old, (g_2 is proved) or
- $f_i \notin s$.Old (we do NOT need to prove $f_i = g_1 \, U \, g_2$).

These together will assure that for each until formula f_i either the right hand side is eventually proved, or that the until formula f_i is not our proof obligation in state s.

Note also that if there are no until formulas, $\mathcal{F} = \emptyset$.

- Many of the emptiness checking algorithms, for example the nested depth first search, do not handle generalized Büchi automata.
- Thus most of the LTL to Büchi translation algorithms make a (non-generalized) Büchi automaton $\mathcal{A}'$ out of the generalized Büchi automaton with the following procedure (which works for any generalized Büchi automaton).
- The construction uses the same “counter trick” as the product construction of two Büchi automata.

Definition 1 Let $\mathcal{A}$ be a generalized Büchi automaton $(\Sigma, S, S^0, \Delta, \mathcal{F})$, where $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$. We now define a (non-generalized) Büchi automaton $\mathcal{A}'$ by case analysis on $\mathcal{F}$:

- if $\mathcal{F} = \emptyset$ then $\mathcal{A}' = (\Sigma, S, S^0, \Delta, S)$,
- if $\mathcal{F} = \{F_1\}$ then $\mathcal{A}' = (\Sigma, S, S^0, \Delta, F_1)$,

- otherwise: $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$, where $n \geq 2$:

  $\mathcal{A}' = (\Sigma, S', S^{0\prime}, \Delta', F_1{}')$, where:
  - $S' = S \times \{1, 2, \ldots, n\}$,
  - $S^{0\prime} = S^0 \times \{1\}$,
  - Δ' is defined as follows: $((s,i), a, (s', j)) \in \Delta'$ iff $(s, a, s') \in \Delta$ and $((s \notin F_i$ and $j = i) \vee (s \in F_i$ and $j = (i\%n) + 1))$, and
  - $F_1{}' = F_1 \times \{1\}$.

Now it holds that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$, and $\mathcal{A}'$ is never smaller than $\mathcal{A}$. In fact, in the worst case $\mathcal{A}'$ has n times as many states as $\mathcal{A}$, where n in the number of acceptance sets. This construction is essentially optimal (result due to H. Tauriainen).

Given an LTL formula f in negation normal form, for the generalized Büchi automaton $\mathcal{A}_f$ a (coarse) upper bound on the number of states it has is $1 + 2^{(2 \cdot |sub(f)|)}$ states. (There is the state "*init*", plus at most as many different states as there are possible combinations of "Old" and "Next" sets, of which there are at most $2^{(2 \cdot |sub(f)|)}$.)

Theorem 2 Given an LTL formula f in negation normal form, the generalized Büchi automaton $\mathcal{A}_f$ has at most $2^{O(|f|)}$ states.

Converting the generalized Büchi automaton to a non-generalized one we get (a coarse) upper bound of $|sub(f)| \cdot (1 + 2^{(2 \cdot |sub(f)|)})$ states. We thus get also the following result.

Theorem 3 Given an LTL formula f in negation normal form, the (non-generalized) Büchi automaton $\mathcal{A}_f$ has at most $2^{O(|f|)}$ states.

There are quite a few highly optimized freely available LTL to Büchi automata translators available. Rolling your own can be educational, but not likely very effective. The exponential blow-up in the construction is unavoidable when translating into Büchi automata. (Recall that it is easy to express a binary counter with n bits with an LTL formula of size $O(n)$.)

Another example of unavoidable blow-up is the LTL formula

$$(\mathit{strong\_fairness}) \rightarrow (\mathit{property})$$

will exhibit this behavior, where

$$\mathit{strong\_fairness} = \bigwedge_{1 \le i \le n} (\Box\Diamond p_i \rightarrow \Box\Diamond q_i).$$

Such fairness assumptions often arise in model checking of liveness properties.