Behavior models and verification

Lecture 4

Jan Kofroň, František Plášil
Basic idea

- Model as a Kripke structure $M$
  - $M$ converted to $A_M$ - Büchi automaton

- Property to be checked as an LTL formula $f$
  - $\neg f$ converted to $A_{\neg f}$ - Büchi automaton

- Product automaton $A_M \times A_{\neg f}$
  - $L(A_M \times A_{\neg f}) = \emptyset$? Yes – fine, i.e. $M \models f$
Recall: Kripke structure for mutex
Recall: Properties of Mutex in LTL

- Mutex specification
  - **Safety:** \( G \neg(c_1 \land c_2) \)
  - **Liveness:** \( G(t_1 \Rightarrow F c_1) \)
  - **Non-blocking:** \( G(n_1 \Rightarrow X t_1) \)
  - **No strict sequencing.** The following should be false at \( s_0 \):
    \[
    G [c_1 \Rightarrow (\neg c_2 \land (c_1 \land \neg c_2) \lor (\neg c_1 \land (\neg c_1 \land c_2) \lor (\neg c_2 \land (\neg c_2 \land c_1) \lor c_2)))]
    \]
    \[
    \land
    \]
    \[
    G [c_2 \Rightarrow (\neg c_1 \land (c_2 \land \neg c_1) \lor (\neg c_2 \land (\neg c_2 \land c_1) \lor c_1))]
    \]
  - **On all paths, all \( p \) precede \( s \ and \ t \)
    \[
    Fp \land Fs \land Ft \land ((\neg s \land \neg t) \lor G \neg p)
    \]
We need to learn

- Büchi automata basics
  - In particular
    - Product $A_1 \times A_2$
    - $L(A) = \emptyset$ ?

- How to covert
  - Kripke $M$ to Buchi $A_M$
  - LTL formula $\neg f$ to Buchi $A_{\neg f}$
Büchi automata

- Büchi automata $A_1$ and $A_2$
  - both over the alphabet $\Sigma = \{a,b\}$
  - Accept infinite sequences (words) over $\Sigma$
Automata on Infinite Words

- Büchi Automata
  - ~ to the definition of a finite state automata
    - Difference: acceptance semantics
  - A (nondeterministic) Büchi automaton $A$ is a tuple $(\Sigma, S, S_0, \Delta, F)$
    - $\Sigma$ is a finite alphabet,
    - $S$ is a finite set of states,
    - $S_0 \subseteq S$ is set of initial states,
    - $\Delta \subseteq S \times \Sigma \times S$ is the transition relation,
    - $F \subseteq S$ is the set of accepting states.
Büchi Automata (cont.)

- \((s, a, s') \in \Delta\) means that there is a transition from state \(s\) to state \(s'\) with symbol \(a\)

- **Definition** of the language accepted by the automaton \(A\) (differs from FSAs):
  - \(A\) accepts a set of infinite words \(L(A) \subseteq \Sigma^\omega\) (the *language* accepted by \(A\)):
    - A *run* \(r\) of \(A\) on an infinite word \(a_0, a_1, \ldots \in \Sigma^\omega\) is an infinite sequence \(s_0, s_1, \ldots\) of states in \(S\), such that \(s_0 \in S^0\), and \((s_i, a_i, s_{i+1}) \in \Delta\) for all \(i \geq 0\)
    - Let \(\text{inf}(r)\) denote the set of states appearing infinitely often in the run \(r\). The run \(r\) is *accepting* if \(\text{inf}(r) \cap F \neq \emptyset\).
    - A word \(w \in \Sigma^\omega\) is accepted by \(A\) if \(A\) has an accepting run on \(w\).
Büchi Automata (cont.)

- \( L(A) \neq 0 \) ?
  - It is easy to check (a linear time algorithm):
    - Iff from some initial state \( s_0 \in S^0 \) an accepting state \( s' \) can be reached, such that \( s' \) can reach itself by a non-empty sequence of transitions.
    - I.e., there should be a path from \( s' \) back to itself via \( \Delta \) which contains at least one edge.
Operations for Büchi Automata

- Defining the Boolean operators on Büchi automata:
  - \( A = A_1 \cup A_2 \)
    - Similar to FSA
  - \( A = A_1 \cap A_2 \)
    - Different from FSA!
  - \( A = \) complement of \( A' \)
    - Technically complex
- Thus also Büchi automata are closed under the Boolean operations
Büchi automata $A_1$ and $A_2$, both over the alphabet $\Sigma = \{a,b\}$. 

Diagram:**

- $A_1$: States $s_0$, $a$, $b$, $s_1$, $b$.
- $A_2$: States $t_0$, $a$, $b$, $t_1$, $b$, $t_2$, $b$. 

Jan Kofroň, František Plášil, Lecture 4
Büchi automaton $A_1 \cup A_2$

$\Sigma = \{a, b\}$
Let $A_1 = (\Sigma, S_1, S_{01}, \Delta_1, F_1)$ and $A_2 = (\Sigma, S_2, S_{02}, \Delta_2, F_2)$ be Büchi automata.

We define the union Büchi automaton to be $(\Sigma, S, S_0, \Delta, F)$, where:

- $S = S_1 \cup S_2$
- $S_0 = S_{01} \cup S_{02}$
- $\Delta = \Delta_1 \cup \Delta_2$
- $F = F_1 \cup F_2$
As a reminder!
However: Different from FSA!
Let $A_1 = (\Sigma, S_1, S_{01}, \Delta_1, F_1)$ and $A_2 = (\Sigma, S_2, S_{02}, \Delta_2, F_2)$ be Büchi automata.

We define the product Büchi automaton to be $(\Sigma, S, S_0, \Delta, F)$, where:

- $S = S_1 \times S_2 \times \{1,2\}$
- $S_0 = S_{01} \times S_{02} \times \{1\}$
- $F = F_1 \times S_2 \times \{1\}$

$\Delta$ as follows
• \( \Delta \)
  - for all \( s, s' \in S_1, t, t' \in S_2, a \in \Sigma, i, j \in \{1,2\} \):
    
    \( ((s, t, i), a, (s', t', j)) \in \Delta \) iff \( (s, a, s') \in \Delta_1 \), \( (t, a, t') \in \Delta_2 \), and:
    - a) \( i = 1, s \in F_1 \), and \( j = 2 \), or
    - b) \( i = 2, t \in F_2 \), and \( j = 1 \), or
    - c) (neither a) or b) above applies and \( j = i \).
Recall: Büchi automata $A_1$ and $A_2$, both over the alphabet $\Sigma = \{a, b\}$.
However: Different from FSA!
Generalized Büchi automaton

- **Just for technical reasons**
  - used in translating LTL formula to Büchi automaton

- **Definition.** A *generalized Büchi automaton* is a tuple
  \[ A_G = (S_0, S, A, R, F_1, \ldots, F_k) \]
  - \( S_0, S, A, R \) are as before
  - each \( F_i \subseteq S \) is a set of accepting states.
  - Accepting runs are defined as before, except that for all \( i \), infinitely many states must be in \( F_i \).

- This is actually not a “generalization”: every generalized automaton can be translated into one of the old kind (by introducing “layers” of accepting states, each to be visited infinitely often)
Definition. Let $AP$ be a set of atomic propositions

- A **Kripke Structure** over $AP$ is a triple $M=(S,R,L)$
  - $S$ is a finite set (of states)
  - $R \subseteq S \times S$ such that for all states $s$, there is a state $s'$ such that $(s,s') \in R$  
    - $R$ is the *transition relation* being left-total -> infinite paths!
  - $L \in S \rightarrow 2^{AP}$  
    - $L$ is the *labeling function* assigning to each state propositions to hold there

Comments

- Write $s \rightarrow s'$ for $(s,s') \in R$.
- Sometimes a Kripke structure will have some *initial states*: $M=(S,S_0,R,L)$.
- $AP \sim$ e.g. Boolean variables
- $M$ is a type of non-deterministic finite state machine
Reminder: Translating mutex ex. to Kripke structure

- $S = \{s0, s1, s2, s3, s4, s5, s6, s7\}$ initial states: $S0 = \{s0\}$
- $R = \{(s0, s1), (s0, s5), (s1, s2), (s1, s3), (s2, s0), \ldots\}$
- $L(s0) = \{n1, n2\}$, $L(s1) = \{t1, n2\}$, $L(s2) = \{c1, n2\}$, \ldots
Reminder: Path in a Kripke structure

- **Definition.** A *path* in a Kripke structure is an *infinite* sequence
  \[ \pi = \pi_0, \pi_1, \pi_2, \ldots \]
  - where for all \( i \geq 1 \), \( \pi_{i-1} \xrightarrow{} \pi_1 \in R \)

- Ex.: \( \pi = s_0, s_5, s_6, s_0, s_5, s_6, s_7, \ldots \)
  - written also:
    \[ \pi = s_0 \rightarrow s_5 \rightarrow s_6 \rightarrow s_0 \rightarrow s_5 \rightarrow s_6 \rightarrow s_7 \ldots \]

- Comments
  - A path \( \pi \) *starts* at a state \( s \) if \( \pi_0 = s \)
  
  - It’s also useful, though not formally necessary, to think about computation *trees.*
Let $M = (S, s^0, R, L)$ be a Kripke structure over a set of atomic propositions $AP$ ($s^0 \in S$).

Define a Büchi automaton

$A_M = (\Sigma, S_M, S_{0M}, \Delta_M, F_M)$

- $\Sigma = 2^{AP}$,
- $S_M = S \cup \{s^i\}$, \text{ ("additional initial symbol")}$
- $S_{0M} = \{s^i\}$,
- $\Delta_M$: For all $s, s' \in S_M, a \in \Sigma : (s, a, s') \in \Delta_M$ iff
  \[ L(s') = a \text{ and } (((s, s') \in R) \text{ or } (s = s^i \text{ and } s' = s^0)); \]
- $F_M = S_M$. \text{ (all states accept)}
From Kripke Structure to Büchi Automaton

- The Büchi automaton $A_M$ accepts exactly those infinite sequences of the labelings which correspond to infinite paths of the Kripke structure starting from some initial state.

- Given an $LTL$ formula $\phi$, we show how to create a Büchi automaton which accepts exactly all the infinite sequences of valuations which satisfy $\phi$. 
Example
Büchi Automaton $A_M$
LTL model checking – basic idea

• Given: M (and a state in M) and f
  ▪ get $A_M$
  ▪ negate the property $f$
  ▪ create a Büchi automaton $A_{\neg f}$.
  ▪ Create $A_M \times A_{\neg f}$ product automaton
    ▪ accepts all traces violating the property $f$
  ▪ Consequently: $M \models f$ iff $L(A_M \times A_{\neg f}) = \emptyset$
Example: LTL Model Checking

- Check whether $M \models f$
  - $f = G (\text{req} \implies (\text{Fack}))$
- By checking for Kripke product automaton
  - $P = A_M \otimes A_{\neg f}$ whether $L(P) = \emptyset$.
  - If so, then $M \models f$, otherwise $M \models \neg f$. 
• Büchi Automaton $A_{\neg f}$
  - $\neg f = F(req \land (G\neg ack))$
  - Recall $f = G (req \Rightarrow (F \neg ack))$
Büchi Automaton $A_M$
• Büchi Automaton $P$
Specific for $F_1 = S_1$ !!!

Let $A_1 = (\Sigma, S_1, S_{01}, \Delta_1, F_1)$ and $A_2 = (\Sigma, S_2, S_{02}, \Delta_2, F_2)$ be Büchi automata. We have defined the product Büchi automaton $A = A_1 \times A_2$, accepting the intersection of $L(A_1)$ and $L(A_2)$.

In the special case $F_1 = S_1$, we can use a simpler product automaton denoted by $A' = A_1 \otimes A_2$:

- $A = (\Sigma, S, S_0, \Delta, F)$
- $S = S_1 \times S_2$
- $S_0 = S_{01} \times S_{02}$
- for all $s, s' \in S_1$, $t, t' \in S_2$, $a \in \Sigma$:
  - $((s, t), a, (s', t')) \in \Delta$ iff $(s, a, s') \in \Delta_1$ and $(t, a, t') \in \Delta_2$
- $F = S_1 \times F_2 = F_1 \times F_2$

Recall that the special case $F_1 = S_1$ occurs whenever $A_1$ is a Kripke structure automaton $A_M$. 

Recall: LTL model checking – basic idea

- Given: $M$ (and a state in $M$) and $f$
  - get $A_M$
  - negate the property $f$
  - create a Büchi automaton $A_{\neg f}$
    - This automaton accepts all violations of the property $f$
  - it holds that $M \models f$ iff $L(A_M \times A_{\neg f}) = \emptyset$

- Key challenge: creating $A_{\neg f}$
Translating LTL into Büchi automaton

- There are several algorithms for translating LTL formulae into Büchi automata
- We show a variant due to Gerth, Peled, Vardi, and Wolper
  - Given an LTL formula $f$, it will generate a Büchi automaton $A_f$ with at most $2^{O(|f|)}$ states
  - The automaton $A_f$ will accept the language $\{w \in \Sigma^\omega : w \models f\}$, where $\Sigma = 2^{AP}$
Translation procedure overview

• Build automaton by starting with an initial node then expanding according to LTL formula.

  ▪ Each node labeled with 3 sets of formulae:
    • New: formulae not yet processed, that may force further expansions.
    • Old: formulae already processed and holding in this state
    • Next: formulae which should hold in any of the next states

• Idea: along any computation path, the New plus Old formulae at any point should hold of the remainder of the path.
  ▪ Construction similar to “tableaux” theorem proving.
Translating LTL into Büchi automaton

- We put a formula $\varphi$ into negation normal form
  - Also called positive normal form
  - All negations appear only in front of atomic propositions

- To avoid the blow-up of the LTL formula, we will use a new operator $R$ (release)
  - $\varphi_1 R \varphi_2 = (G \varphi_2) \lor (\varphi_2 U (\varphi_1 \land \varphi_2))$
  - Informally: $\varphi_1$ “releases” $\varphi_2$
A simplification

• Before constructing the automata, put the LTL formula in *negation normal form*:
  - mentions only AP elements, $\neg$, $\land$, $\lor$, $X$, $R$, $U$;
  - and all $\neg$’s only in front of AP elements
• Repeatedly rewrite using the LTL equivalences:
  - $F \, g = true \, U \, g$
  - $G \, g = false \, R \, g$
  - $\neg(f \, U \, g) = \neg f \, R \, \neg g$
  - $\neg(f \, R \, g) = \neg f \, U \, \neg g$
  - and Boolean equivalences (De Morgan)
Translating LTL into Büchi automaton

- Translation into negation normal form
  - $\neg X \varphi_1 = X \neg \varphi_1$
  - $\neg (\varphi_1 \cup \varphi_2) = \neg \varphi_1 R \neg \varphi_2$
  - $\neg (\varphi_1 R \varphi_2) = \neg \varphi_1 \cup \neg \varphi_2$

- Bottom line
  - Negations should be only in front of APs
Translating LTL into Büchi automaton

- The algorithm for the LTL formula $\rightarrow$ Büchi automata translation
  - The algorithm will manipulate a data structure called *node* during its run
    - ID: A unique identifier of a node (a number)
    - Incoming: A list of node IDs
    - Old $\subseteq$ sub(f)
    - New $\subseteq$ sub(f)
    - 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*
Technical: To implement the algorithm, the following functions are defined

- $\text{Neg}(\text{true}) = \text{false}$
- $\text{Neg}(\text{false}) = \text{true}$
- $\text{Neg}(p) = \neg p, p \in \text{AP}$
- $\text{Neg}(\neg p) = p, p \in \text{AP}$
The functions $\text{New}1(f)$, $\text{Next}1(f)$, and $\text{New}2(f)$ are tabulated below. They match the recursive definitions for disjunction, until, release, and next-time:

<table>
<thead>
<tr>
<th>$f$</th>
<th>$\text{New}1(f)$</th>
<th>$\text{Next}1(f)$</th>
<th>$\text{New}2(f)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1 \lor f_2$</td>
<td>${f_2}$</td>
<td>$\emptyset$</td>
<td>${f_1}$</td>
</tr>
<tr>
<td>$f_1 U f_2$</td>
<td>${f_1}$</td>
<td>${f_1 U f_2}$</td>
<td>${f_2}$</td>
</tr>
<tr>
<td>$f_1 R f_2$</td>
<td>${f_2}$</td>
<td>${f_1 R f_2}$</td>
<td>${f_1, f_2}$</td>
</tr>
<tr>
<td>$X f_1$</td>
<td>$\emptyset$</td>
<td>${f_1}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Translating LTL into Büchi automaton

- Basic idea of the LTL formula $\rightarrow$ Büchi automata translation
  - To prove that $w \models \varphi_1 \lor \varphi_2$ it suffices to either prove that
    - $w \models \varphi_1$, or
    - $w \models \varphi_2$
  - To prove that $w \models \varphi_1 \lor \varphi_2$ it suffices to either prove that
    - $w \models \varphi_2$, or
    - $w \models \varphi_1$ and $w \models X(\varphi_1 \lor \varphi_2)$
    - The only restriction is that the second case can only be used infinitely often iff the first case is also used infinitely often (we will use Büchi acceptance sets to handle that)
  - To prove that $w \models \varphi_1 R \varphi_2$ it suffices to either prove that
    - $w \models \varphi_1$ and $w \models \varphi_2$, or
    - $w \models \varphi_2$ and $w \models X(\varphi_1 R \varphi_2)$
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 := 0; // 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 = 0;
    node.Next = 0;
    expand(node); // Call the recursive expand procedure
    return nodes;
end procedure
Algorithm 2  \textit{LTL to Büchi translation main loop}

**procedure** expand(node: Node)

\textbf{local} node, node1, node2: Node;

\textbf{local} f: Formula;

\textbf{if} node.New = \emptyset \textbf{then}

\textbf{if} \exists \text{ node1} \in \text{ nodes with node1.Old} = \text{ node.Old} \land \text{ node1.Next} = \text{ node.Next} \textbf{then}

\hspace{1em} node1.Incoming := node1.Incoming \cup \text{ node.Incoming}; \text{ // redirect arcs to “node1”}

\hspace{1em} \textbf{return}; \text{ // Discard “node” by not storing it to “nodes”}

\textbf{else}

\hspace{1em} nodes := nodes \cup \{ \text{ node} \}; \text{ // “node” is ready, add it to the automaton}

\hspace{1em} node2 := \text{ NewNode}(); \text{ // Create “node2” to prove formulas in “node.Next”}

\hspace{1em} node2.ID := \text{ GetID}();

\hspace{1em} node2.Incoming := \{ \text{ node.ID} \};

\hspace{1em} node2.New = \{ \text{ node.Next} \};

\hspace{1em} node2.Old = \emptyset;

\hspace{1em} node2.Next = \emptyset;

\hspace{1em} expand(node2);

\hspace{1em} \textbf{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) ∈ 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 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 \( f_1 \land f_2 \)
f2 := right(f); // Obtain subformula “f2” from right side of \( f_1 \land f_2 \)
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: $f_1 \lor f_2, f_1 U f_2, f_1 R f_2$

      // 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 (NewI(f) \setminus node.Old); // Prove things in NewI(f)
      node1.Old := node.Old \cup \{ f \}; // Recall that “node1” node proves “f”
      node1.Next = node.Next \cup NextI(f); // On the next time, prove things in NextI(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 $Xf_1$
    
    // This one handles the case $Xf_1$
    node.Old := node.Old $\cup \{ f \}$; // Recall that “node” node proves “f”
    node.Next = node.Next $\cup Next1(f)$; // On the next time, prove things in $Next1(f)$
    expand(node); // Handle the rest of the formulas in “node.New”
    return;
end procedure
When expansion is completed, to get a Büchi automaton we need to add two things:

- Transition labels: for each edge \((u,v)\), add a label \(X\) for each set \(X\) of variables making all the literals in \(Old(v)\) true.

- Accepting states: Consider an arbitrary path, and suppose \(p \lor q\) is true at the start. By construction,
  - either \(q\) is labeled true at some point,
  - or \(p\) and \(p \lor q\) both labeled true forever.
    - Need to rule out the second case
  - Note: for formulas not containing \(U\), the set of accepting states is empty \(\rightarrow\) in automaton any infinite paths are accepting
Accepting states

- For each subformula (of the original formula) of the form $f \cup g$, add a set $F_{f \cup g}$ of accepting states: put $u$ in $F_{f \cup g}$ iff either
  - $g \in \text{Old}(u)$,
  - or
  - $f \cup g \notin \text{Old}(u)$.

- **Recall:** A path (run) is accepted iff for each accepting set $F_i$, infinitely many states (nodes) in the path are in $F_i$.

- **Observe:** A path is NOT accepted iff there is some subformula $f \cup g$ such that at some point in the path, $f \cup g$ becomes continuously true, but $g$ never becomes true.
Translating LTL into Büchi automaton: Example

Let’s trace through the algorithm with the following input:

\[ a \mathcal{U} (p \land q) \]
Simple observation: Validity of formula depends on validity of primitive variables → we need to decompose the formula to isolate those variables and introduce corresponding arcs.
Translating LTL into Büchi automaton

- Recursive construction of a transition system (Büchi automaton)
  - Three set at each node:
    - *New* – members to be processed within this node
    - *Old* – members already processed within this node (proposition that holds at this node)
    - *Next* – new members of the next node
The functions $\text{New1}(f)$, $\text{Next1}(f)$, and $\text{New2}(f)$ are tabulated below. They match the recursive definitions for disjunction, until, release, and next-time:

<table>
<thead>
<tr>
<th>$f$</th>
<th>$\text{New1}(f)$</th>
<th>$\text{Next1}(f)$</th>
<th>$\text{New2}(f)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1 \lor f_2$</td>
<td>${f_2}$</td>
<td>$\emptyset$</td>
<td>${f_1}$</td>
</tr>
<tr>
<td>$f_1 U f_2$</td>
<td>${f_1}$</td>
<td>${f_1 U f_2}$</td>
<td>${f_2}$</td>
</tr>
<tr>
<td>$f_1 R f_2$</td>
<td>${f_2}$</td>
<td>${f_1 R f_2}$</td>
<td>${f_1, f_2}$</td>
</tr>
<tr>
<td>$X f_1$</td>
<td>$\emptyset$</td>
<td>${f_1}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
• Select a formula from the NEW set and process it in suitable way.
• The only formula here is
  \[ a \lor (p \land q) \]
  \[ \rightarrow \text{create 2 nodes} \]
\[ N_1.\text{NEW} = \{ a \cup ( p \land q ) \} \]

\[ N_1.\text{OLD} = 0; \]

\[ N_1.\text{NEXT} = 0; \]
\[ N_2.\text{NEW} = \{ a \} \]
\[ N_2.\text{OLD} = \{ a \cup (p \land q) \} \]
\[ N_2.\text{NEXT} = \{ a \cup (p \land q) \} \]
\[ N_3.\text{NEW} = \{ p \land q \} \]
\[ N_3.\text{OLD} = \{ a \cup (p \land q) \} \]
\[ N_3.\text{NEXT} = 0 \]
$N_2.\text{NEW} = 0$

$N_2.\text{OLD} = \{ a \cup ( p \land q ) ; a \}$

$N_2.\text{NEXT} = \{ a \cup ( p \land q ) \}$
\[ N_4.\text{NEW} = \{ \, a \lor (p \land q) \, \} \]
\[ N_4.\text{OLD} = 0 \]
\[ N_4.\text{NEXT} = 0 \]

• This is similar to starting at \( N_1 \) thus \( N_4 \) will be replaced by two nodes
..skip some steps…

\[ N_5.\text{NEW} = 0 \]
\[ N_5.\text{OLD} = \{ a \cup (p \land q) ; a \} \]
\[ N_5.\text{NEXT} = \{ a \cup (p \land q) \} \]

\[ N_6.\text{NEW} = \{ p \land q \} \]
\[ N_6.\text{OLD} = \{ a \cup (p \land q) \} \]
\[ N_6.\text{NEXT} = 0 \]
Since $N_5$ is “equivalent” to $N_2$, discard the $N_5$ node and redirect edges to $N_2$
$N_6$.NEW = \{ p ; q \} \\
$N_6$.OLD = \{ a U ( p \land q ) ; p \land q \} \\
$N_6$.NEXT = 0

$N_6$.NEW = 0 \\
$N_6$.OLD = \{ a U ( p \land q ) ; p \land q ; p ; q \} \\
$N_6$.NEXT = 0
\text{N}_7.\text{NEW} = 0
\text{N}_7.\text{OLD} = 0
\text{N}_7.\text{NEXT} = 0
\( N_8.\text{NEW} = 0 \)
\( N_8.\text{OLD} = 0 \)
\( N_8.\text{NEXT} = 0 \)
Since \( N_8 \) is “equivalent” to \( N_7 \), discard the \( N_8 \) node and redirect edges to \( N_7 \)
Backtrack and process node $N_3$...

$N_3\.NEW = 0$

$N_3\.OLD = \{ a \cup ( p \land q ) ; p \land q ; p ; q \}$

$N_3\.NEXT = 0$
Since $N_3$ is “equivalent” to $N_6$, discard the $N_3$ node and redirect edges to $N_6$

\[ N_2.OLD = \{ a \cup (p \land q); a \} \]
\[ N_6.OLD = \{ a \cup (p \land q); p \land q; p; q \} \]
\[ N_7.OLD = 0 \]
When expansion is completed, to get a Büchi automaton we need to add two things:

- Transition labels: for each edge \((u,v)\), add a label \(X\) for each set \(X\) of variables making all the literals in \(\text{Old}(v)\) true.

- Accepting states: Consider an arbitrary path, and suppose \(p U q\) is true at the start. By construction,
  - either \(q\) is labeled true at some point,
  - or \(p\) and \(p U q\) both labeled true forever.
  - Need to rule out the second case.

Note: for formulas not containing \(U\), the set of accepting states is empty -> in automaton any infinite paths are accepting.
Accepting states

- For each subformula (of the original formula) of the form $f U g$, add a set $F_{f U g}$ of accepting states: put $u$ in $F_{f U g}$ iff either
  - $g \in \text{Old}(u)$,
  - or $f U g \notin \text{Old}(u)$.

- **Recall**: A path (run) is accepted iff for each accepting set $F_i$, infinitely many states (nodes) in the path are in $F_i$.

- **Observe**: A path is NOT accepted iff there is some subformula $f U g$ such that at some point in the path, $f U g$ becomes continuously true, but $g$ never becomes true.
Accepting states and transitions

\[ a \cup (p \land q) \]