Explicit LTL model checking

- Markov chains
- Timed automata
- Labelled transition system
- Kripke structure

Model:
- open
- close
- heat
- start
- empty

Property specification:

\[ AG(\text{start} \rightarrow AF \text{heat}) \]

Model checker:
- Property satisfied
- Property violated

Error report
Example
Büchi Automaton $A_M$
LTL model checking – basic idea

Given: $M$ (and a state in $M$) and $f$

1. create $A_M$
2. negate the property $f$
3. create a Büchi automaton $A_{\neg f}$.
   - This automaton accepts all violations of the property $f$
4. it holds that $M \models f$ iff $L(A_M \times A_{\neg f}) = \emptyset$
Example: LTL Model Checking

- Check whether $M \models f$
  - $f = G(req \rightarrow (F \text{ack}))$
- By checking product automaton $P$, whether $L(P) = \emptyset$
  - $P = A_M \times A_{\neg f}$
  - 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 $f = G (req \rightarrow (F ack))$
Büchi product automaton $P$
Kripke Product Automaton Corner Case

- **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 $A_M$
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 $O(2^{|f|})$ 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:
  
  - $F g = \text{true } U g$
  - $G g = \text{false } R g$
  - $\neg(f U g) = \neg f R \neg g$
  - $\neg(f R g) = \neg f U \neg g$
  - and Boolean equivalences (De Morgan)
Translation into negation normal form

- $\neg X \varphi_1 = X \neg \varphi_1$
- $\neg (\varphi_1 U \varphi_2) = \neg \varphi_1 R \neg \varphi_2$
- $\neg (\varphi_1 R \varphi_2) = \neg \varphi_1 U \neg \varphi_2$
The algorithm for the LTL formula $\rightarrow$ Büchi automata translation

- The algorithm will manipulate a date 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 $NewI(f)$, $NextI(f)$, and $New2(f)$ are tabulated below. They match the recursive definitions for disjunction, until, release, and next-time:

<table>
<thead>
<tr>
<th>$f$</th>
<th>$NewI(f)$</th>
<th>$NextI(f)$</th>
<th>$New2(f)$</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>
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
    - $w \models \varphi_1 \text{ and } w \models \varphi_2$, or
    - $w \models \varphi_2 \text{ and } w \models X(\varphi_1 R \varphi_2)$
**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 \textit{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}

\textbf{procedure} expand(node: Node)
\hspace{1em} \textbf{local} node, node1, node2: Node;
\hspace{1em} \textbf{local} f: Formula;
\hspace{1em} \textbf{if} node.New $= 0$ \textbf{then}
\hspace{2em} \textbf{if} \exists \text{ node1} \in \text{nodes with node1.Old} = \text{node.Old} \land \text{node1.Next} = \text{node.Next} \textbf{then}
\hspace{3em} \text{node1.Incoming} := \text{node1.Incoming} \cup \text{node.Incoming}; \hspace{0.5em} // \text{redirect arcs to \textquote{node1}''}
\hspace{3em} \textbf{return}; \hspace{0.5em} // \text{Discard \textquote{node}'' by not storing it to \textquote{nodes}''}
\hspace{2em} \textbf{else}
\hspace{3em} \text{nodes} := \text{nodes} \cup \{ \text{ node } \}; \hspace{0.5em} // \text{\textquote{node}'' is ready, add it to the automaton}
\hspace{3em} \text{node2} := \text{NewNode()}; \hspace{0.5em} // \text{Create \textquote{node2}'' to prove formulas in \textquote{node.Next}''}
\hspace{3em} \text{node2.ID} := \text{GetID()};
\hspace{3em} \text{node2.Incoming} := \{ \text{node.ID} \};
\hspace{3em} \text{node2.New} = \{ \text{node.Next} \};
\hspace{3em} \text{node2.Old} = \emptyset;
\hspace{3em} \text{node2.Next} = \emptyset;
\hspace{3em} \text{expand(node2)};
\hspace{3em} \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  \textit{Expanding simple formulas}

\begin{verbatim}
procedure expand_simple(node: Node, f: Formula)
    if f = false or \texttt{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
\end{verbatim}
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: \( f_1 \lor f_2, f_1 \cup f_2, f_1 \Rightarrow f_2 \)

  // 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 \cup (New2(f) \setminus node.Old); // Prove things in New2(f)
node2.Old := node.Old \cup \{ 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 ∪ { f }; // Recall that “node” node proves “f”
  node.Next = node.Next ∪ 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
Post – expansion

- 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 \(p \cup q\) is true at the start. By construction,
    - either \(q\) is labeled true at some point,
    - or \(p\) and \(p \cup q\) both labeled true forever.
      - Need to rule out the second case
    - Note: for formulas not containing \(U\), all infinite paths are accepting
Accepting states

- For each subformula (of the original formula) of the form $pUq$, add a set $F_{pUq}$ of accepting states: put $s$ in $F_{pUq}$ iff either
  - $q \in \text{Old}(s)$,
  - or
  - $pUq \notin \text{Old}(s)$

- **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 $pUq$ such that at some point in the path, $pUq$ becomes continuously true, but $q$ never becomes true
Example

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

$$\alpha \cup (p \land q)$$
\[ a \cup (p \land q) \]
\[ N_1.\text{NEW} = \{ a \cup ( p \land q ) \} \]

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

\[ N_1.\text{NEXT} = 0; \]
- Select a formula from the NEW set and process it in suitable way.
- The only formula here is
  \[ a \cup ( p \land q ) \]
  \[ \rightarrow \text{create 2 nodes} \]
\[ N_2.\text{NEW} = \{ a \} \]
\[ N_2.\text{OLD} = \{ a \cup (p \land q) \} \]
\[ N_2.\text{NEXT} = \{ a \cup (p \land q) \} \]

\[ N_3.\text{NEW} = \{ p \land q \} \]
\[ N_3.\text{OLD} = \{ a \cup (p \land q) \} \]
\[ N_3.\text{NEXT} = 0 \]
\[ N_2.\text{NEW} = 0 \]
\[ N_2.\text{OLD} = \{ a \cup ( p \land q ) ; a \} \]
\[ N_2.\text{NEXT} = \{ a \cup ( p \land q ) \} \]
\( N_4.\text{NEW} = \{ a \cup (p \land q) \} \)
\( N_4.\text{OLD} = 0 \)
\( N_4.\text{NEXT} = 0 \)
..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 $N_2$. 
\[ N_6.\text{NEW} = \{ p ; q \} \]
\[ N_6.\text{OLD} = \{ a \cup ( p \land q ) ; p \land q \} \]
\[ N_6.\text{NEXT} = 0 \]

\[ N_6.\text{NEW} = 0 \]
\[ N_6.\text{OLD} = \{ a \cup ( p \land q ) ; p \land q ; p ; q \} \]
\[ N_6.\text{NEXT} = 0 \]
N_7.NEW = 0
N_7.OLD = 0
N_7.NEXT = 0
\text{N}_8.\text{NEW} = 0
\text{N}_8.\text{OLD} = 0
\text{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 $N_6$

$N_2.OLD = \{ a \cup (p \land q) ; a \}$
$N_6.OLD = \{ a \cup (p \land q) ; p \land q ; p ; q \}$
$N_7.OLD = 0$
Accepting states and transitions

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