Mutual Exclusion Again

\( \text{st} \) — status of the process (critical section not, or trying)

\( \text{other-st} \) — status of the other process

\( \text{turn} \) — ensures that they take turns

MODULE main

VAR

pr1 : process prc(pr2.st, turn, 0);
pr2 : process prc(pr1.st, turn, 1);

ASSIGN

init(turn) := 0;

-- safety

SPEC AG!((pr1.st = c) & (pr2.st = c))

-- liveness

SPEC AG((pr1.st = t) -> AF (pr1.st = c))

SPEC AG((pr2.st = t) -> AF (pr2.st = c))

-- no strict sequencing

SPEC EF(pr1.st = c & E[pr1.st = c U (!pr1.st = c &

E[! pr2.st = c U pr1.st = c ])]))

Model (Cont'd)

MODULE prc(other-st, turn, myturn)

VAR

st : \{n, t, c\};

ASSIGN

init(st) := n;

next(st) :=

case

(st = n) : \{t, n\};

(st = t) & (other-st = n) : c;

(st = t) & (other-st = t) & (turn = myturn) : c;

(st = c) : \{c, n\};

1 : st;

esac;

next(turn) :=

case

turn = myturn & st = c : !turn;

1 : turn;

esac;

FAIRNESS running

FAIRNESS !(st = c)
Comments:

- The labels in the slide above denote the process which can make the move.

- Variable turn was used to differentiate between states $s_3$ and $s_9$, so we now distinguish between $ct0$ and $ct1$. But transitions out of them are the same.

- Removed the assumption that the system moves on each tick of the clock. So, the process can get stuck, and thus the fairness constraint.

- In general, what is the difference between the single fairness constraint $\psi_1 \land \psi_2 \land \ldots \land \psi_n$ and $n$ fairness constraints $\psi_1, \psi_2$, etc., written on separate lines under FAIRNESS?
Notion of Fairness

*Fairness:* a path $p$ is fair w.r.t. property $\psi$ if $\psi$ is true on $p$ infinitely often.

We may want to evaluate A and E constraints only over those paths.

Example: each process will run infinitely often; a process can stay in a critical section arbitrarily long, as long as it eventually leaves.

Two types of fairness: simple
  *Property $\phi$ is true infinitely often.*

and compound
  *If $\phi$ is true infinitely often, then $\psi$ is also true infinitely often.*

SMV can deal only with simple fairness.

Formal Definition of Fairness

Let $C = \{\psi_1, \psi_2, ..., \psi_n\}$ be a set of $n$ fairness constraints. A computation path $s_0, s_1, ...$ is fair w.r.t. $C$ if for each $i$ there are infinitely many $j$ s.t. $s_j \models \psi_i$, that is, each $\psi_i$ is true infinitely often along the path.

We use $A_C$ and $E_C$ for the operators A and E restricted to fair paths.

$E_C U$, $E_C G$ and $E_C X$ form an adequate set.

Also, a path is fair iff any suffix of it is fair. Finally,

$$E_C[\phi U \psi] = E[\phi U (\psi \land E_C G T)]$$

$$E_C X \phi = EX (\phi \land E_C G T)$$

We only need an algorithm for $E_C G \phi$. 
Algorithm for $\text{ECG}\phi$

- Restrict the graph to states satisfying $\psi$; of the resulting graph, we want to know from which states there is a fair path.
- Find the maximal strongly connected components (SCCs) of the restricted graph;
- Remove an SCC if, for some $\psi_i$, it does not contain a state satisfying $\psi_i$. The resulting SCCs are the fair SCCs. Any state of the restricted graph that can reach one has a fair path from it.
- Use backwards breadth-first searching to find the states on the restricted graph that can reach a fair SCC.

Complexity: $O(n \times |f| \times (S + |R|))$ (still linear in the size of the model and formula).

Guidelines for Modeling with SMV

- Identify inputs from the environment.
- Make sure that the environment is non-deterministic. All constraints on the environment should be carefully justified.
- Determine the states of the system and its outputs. Model them in terms of the environmental inputs.
- Specify fairness criteria, if any. Justify each criterion. Remember that you can over-specify the system. Fairness may not be implementable, and in fact may result in no behaviors.
- Specify correctness in CTL. Comment each CTL property in English.
- Ensure that CTL properties are not satisfied vacuously. That is, each universally-quantified property should be paired up with an existentially-specified property. Also check that LHSs of implications are not always false.

Examples:
- $\text{AG} (a) \longrightarrow$
- $\text{AG} (a \rightarrow b) \longrightarrow$
Binary Decision Diagrams

- Representation of Boolean Functions
- BDDs, OBDDs, ROBDDs
- Operations
- Model-Checking over BDDs

Readings: 6.1-6.3 of Huth, Ryan

Boolean Functions

Boolean functions: \( B = \{0,1\} \), 
\[ f : B \times \cdots \times B \rightarrow B \]

Boolean expressions:
\[ t ::= x \mid 0 \mid 1 \mid \neg t \mid t \land t \mid t \lor t \mid t \rightarrow t \mid t \leftarrow t \]

Truth assignments: \( \rho \), 
\[ [v_1 x_1, v_2 x_2, \cdots, v_n x_n] \]

Satisfiable: Exists \( \rho \) s.t. \( t[\rho] = 1 \)

Tautology: Forall \( \rho \), \( t[\rho] = 1 \)
Truth Tables

<table>
<thead>
<tr>
<th>xyz</th>
<th>x → y, z</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
</tr>
</tbody>
</table>

$2^n$ entries

What is a good representation of boolean functions?

Perfect representation is hopeless:

**Theorem 1** (Cook's Theorem)
Satisfiability of Boolean expressions is NP-complete.

(Tautology-checking is co-NP-complete)

Good representations are
compact and
efficient
on real-life examples
Combinatorial circuits

Shannon Expansion

Def: $x \rightarrow y_0, y_1 = (x \land y_0) \lor (\neg x \land y_1)$

$x$ is the test expression and thus this is an if-then-else.

We can represent all operators using if-then-else on unnegated variables and constants 0(false) and 1(true). This is called INF.

Shannon expansion w.r.t. $x$:
$$t = x \rightarrow t[1/x], t[0/x]$$

Any boolean expression is equivalent to an expression in INF.

Are these equivalent? Do they represent a tautology? Are they satisfiable?
**Example**

\[ t = (x_1 \leftrightarrow y_1) \land (x_2 \leftrightarrow y_2) \]. Represent this in
INF form with order \( x_1, y_1, x_2, y_2 \).

\[ t = x_1 \rightarrow t_1, t_0 \]
\[ t_0 = y_1 \rightarrow 0, t_{00} \]
  (since \( x_1 = 1, y_1 = 0 \rightarrow t = 0 \))
\[ t_1 = y_1 \rightarrow t_{11}, 0 \]
  (since \( x_1 = 0, y_1 = 1 \rightarrow t = 0 \))
\[ t_{00} = x_2 \rightarrow t_{001}, t_{000} \]
\[ t_{11} = x_2 \rightarrow t_{111}, t_{000} \]
\[ t_{000} = y_2 \rightarrow 0, 1 \]  \( (x_1 = 0, y_1 = 0, x_2 = 0) \)
\[ t_{001} = y_2 \rightarrow 1, 0 \]  \( (x_1 = 0, y_1 = 0, x_2 = 1) \)
\[ t_{110} = y_2 \rightarrow 0, 1 \]  \( (x_1 = 1, y_1 = 1, x_2 = 0) \)
\[ t_{111} = y_2 \rightarrow 1, 0 \]  \( (x_1 = 1, y_1 = 1, x_2 = 1) \)

Lots of common subexpressions:
- identify them!

BDDs – directed acyclic graph of Boolean expressions. If the variables occur in the same
ordering on all paths from root to leaves, we
call this OBDD.
Example OBDD

OBDD for \((x_1 \leftrightarrow y_1) \land (x_2 \leftrightarrow y_2)\) with ordering \(x_1 < y_1 < x_2 < y_2\)

If an OBDD does not contain any redundant tests, it is called ROBDD.

ROBDDs

A *Binary Decision Diagram* is a rooted, directed, acyclic graph \((V, E)\). \(V\) contains (up to) two terminal vertices, \(0, 1 \in V\). \(v \in V \setminus \{0, 1\}\) are non-terminal and have attributes \(\text{var}(v)\), and \(\text{low}(v), \text{high}(v) \in V\).

A BDD is *ordered* if on all paths from the root the variables respect a given total order.

A BDD is *reduced* if for all non-terminal vertices \(u, v\),

1) \(\text{low}(u) \neq \text{high}(u)\)

2) \(\text{low}(u) = \text{low}(v), \text{high}(u) = \text{high}(v), \text{var}(u) = \text{var}(v)\) implies \(u = v\).
ROBDD Examples

Canonicity of ROBDDs

Lemma 1 (Canonicity lemma) For any function $f : B^n \rightarrow B$ there is exactly one ROBDD $b$ with variables $x_1 < x_2 < \cdots < x_n$ such that $t_b[v_1/x_1, \cdots, v_n/x_n] = f(v_1, \cdots, v_n)$ for all $(v_1, \ldots, v_n) \in B^n$.

Consequences:
- $b$ is a tautology if and only if $b = 1$
- $b$ is satisfiable if and only if $b \neq 0$
But.

The size of ROBDD depends *significantly* on the chosen variable ordering!

Example: ROBDD for \((x_1 \Leftrightarrow y_1) \land (x_2 \Leftrightarrow y_2)\) with ordering \(x_1 < x_2 < y_1 < y_2\)

Under ordering \(x_1 < y_1 < x_2 < y_2\) had 6 nodes.

Furthermore...

- The size according to one ordering may be exponentially smaller than another ordering.
- Figuring out the optimal ordering of variables is co-NP-complete.
- Some functions have small size independent of ordering, e.g., parity.
- Some functions have large size independent of ordering, e.g., multiplication
### Implementing BDDs

{root: integer; var, low, high: array of integer;}

<table>
<thead>
<tr>
<th>var</th>
<th>low</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

### Helper Functions: Makenode and Hashing

Makenode ensures reducedness using a hash table

\[ H : (i, l, h) \rightarrow u \]

Makenode\((H, max, b, i, l, h)\)

1:  \textbf{if} \ l = h \ \textbf{then return} \ l
2:  \textbf{else if} member\((H, i, l, h)\)
3:  \textbf{then return} lookup\((H, i, l, h)\)
4:  \textbf{else} max \leftarrow max + 1
5:  b.var\(max) \leftarrow i
6:  b.low\(max) \leftarrow l
7:  b.high\(max) \leftarrow h
8:  insert\((H, i, l, h, max)\)
9:  \textbf{return} max
**Build**

Build: Maps a Boolean expression $t$ into an ROBDD.

**function** Build$(t)$
1:    $H \leftarrow \text{emptytable}; max \leftarrow 1$
2:    $b.\text{root} \leftarrow \text{build}'(t, 1)$
3:    return $b$

**function** build'$(t, i)$
1:    if $i > n$ then
2:        if $t$ is false then return 0
3:            else return 1
4:    else $l \leftarrow \text{build}'(t[0/x_i], i + 1)$
5:        $h \leftarrow \text{build}'(t[1/x_i], i + 1)$
6:    return makenode$(H, max, b, i, l, h)$
**Boolean Operations on ROBDDs**

Ordering: \( x_1 < \cdots < x_n \)

\[
(x_i \rightarrow l_1, l_0) \op (x_i \rightarrow h_1, h_0) = x_i \rightarrow (l_1 \op h_1), (l_0 \op h_0)
\]

---

**Boolean Operations on ROBDDs (Cont’d)**

If \( x_i < x_j \):

\[
(x_i \rightarrow l_1, l_0) \op (x_j \rightarrow h_1, h_0) = x_i \rightarrow (l_1 \op (x_j \rightarrow h_1, h_0)),
\]

\[
(l_0 \op (x_j \rightarrow h_1, h_0)) \op
\]

---

65

---

66
**Function** Apply

Used to perform operations on two ROBDDs.

Example:

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>x1</td>
<td>x2</td>
<td>x2</td>
</tr>
<tr>
<td>x2</td>
<td>x2</td>
<td>x3</td>
<td>x3</td>
</tr>
<tr>
<td>x3</td>
<td>x4</td>
<td>x4</td>
<td></td>
</tr>
</tbody>
</table>
```

Can be either recursive or using dynamic programming.

**Other Operations on ROBDDs**

*Restrict* \( b[v/x] \)
given a truth assignment for \( x \), compute ROBDD for \( b \)

*Size* \( \text{size}(b) = |\{ \rho \mid b[\rho] = 1 \}| \)
"number of valid truth assignments"

*Anysat* \( \text{anysat}(b) = \rho \), for some \( \rho \) with \( b[\rho] = 1 \)
"give a satisfying assignment"

*Compose* \( \text{compose}(b, x, b') = b[x/b'] \)
"substitute \( b' \) for all free occurrences of \( x \)"

*Existential quantification* \( \exists x.b = b[x/0] \lor b[x/1] \)

Using dynamic hash-table implementation, can get amortized cost for operations to be \( O(1) \).
## Representing Boolean Functions

<table>
<thead>
<tr>
<th>Representation of boolean functions</th>
<th>compact?</th>
<th>satisfy</th>
<th>validity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prop. formulas</td>
<td>often</td>
<td>hard</td>
<td>hard</td>
</tr>
<tr>
<td>Formulas in DNF</td>
<td>sometimes</td>
<td>easy</td>
<td>hard</td>
</tr>
<tr>
<td>Formulas in CNF</td>
<td>sometimes</td>
<td>hard</td>
<td>easy</td>
</tr>
<tr>
<td>Ordered truth tables</td>
<td>never</td>
<td>hard</td>
<td>hard</td>
</tr>
<tr>
<td>Reduced OBDDs</td>
<td>often</td>
<td>easy</td>
<td>easy</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Representation of boolean functions</th>
<th>(\land)</th>
<th>(\lor)</th>
<th>(\neg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prop. formulas</td>
<td>easy</td>
<td>easy</td>
<td>easy</td>
</tr>
<tr>
<td>Formulas in DNF</td>
<td>hard</td>
<td>easy</td>
<td>hard</td>
</tr>
<tr>
<td>Formulas in CNF</td>
<td>easy</td>
<td>hard</td>
<td>hard</td>
</tr>
<tr>
<td>Ordered truth tables</td>
<td>hard</td>
<td>hard</td>
<td>hard</td>
</tr>
<tr>
<td>Reduced OBDDs</td>
<td>medium</td>
<td>medium</td>
<td>easy</td>
</tr>
</tbody>
</table>

## Uses of ROBDDs

Symbolic reasoning about:
- Combinatorial circuits
- Sequential circuits
- Automata
- Program analysis (theorem-proving)

and

- Temporal logic model checking