Asynchronous Circuit Design

Chris J. Myers

Lecture 4: Graphical Representations
Chapter 4
HDL’s allow specification of large systems.
Graphs allow pictorial representation of small examples, and they are used by virtually every CAD algorithm.

The chapter discusses the following types of graphs:
- State machines
- Petri-nets
A graph $G$ is composed of a finite nonempty set of vertices $V$ and a binary relation, $R$ ($R \subseteq V \times V$).

**Undirected graphs:**
- $R$ is an irreflexive symmetric relation.
- Since $R$ is symmetric, $(u, v) \in R \Rightarrow (v, u) \in R$.
- $E$ is the set of symmetric pairs, or *edges* (denoted $uv$).

**Directed graphs, or digraphs:**
- $R$ does not need to be either irreflexive or symmetric.
- $E$ is the set of *directed edges* or *arcs* (denoted $(u,v)$).
A Simple Graph

\[\begin{align*}
v1 & \quad v2 \\
& \quad | \\
& \quad v4 \\
& \quad | \\
v5 & \quad v3
\end{align*}\]
A Simple Directed Graph

![Graph Diagram]

- $v_1$
- $v_2$
- $v_3$
- $v_4$
- $v_5$
Additional Graph Definitions

- $|V|$ is called the **order** of $G$.
- $|E|$ is called the **size** of $G$.
- $V(G)$ and $E(G)$ are the vertex and edge sets for $G$.
- If $e = (u, v) \in E(G)$, $e$ joins $u$ and $v$.
- If $e = (u, v) \in E(G)$, $u$ and $v$ are **incident** with $e$.
- If $(u, v) \in E(G)$, $v$ is **adjacent** to $u$.
- If $(u, v) \not\in E(G)$, $u$ and $v$ are **nonadjacent** vertices.
- $u$-$v$ path is an alternating sequence of vertices and edges beginning with $u$ and ending with $v$.
- The length of a $u$-$v$ path is the number of edges in the path.
- If there exists a $u$-$v$ path, then $v$ is reachable from $u$.
- A $u$-$v$ path is simple if it does not repeat any vertex.
- If for every pair of vertices $u$ and $v$ there exists a $u$-$v$ path, the graph is connected.
A Unconnected Graph

```plaintext
v1
  / \ \\
v2   v3
```

v4

v5
In a digraph, a \( u \)-\( v \) path forms a *cycle* if \( u = v \).

If the \( u \)-\( v \) path excluding \( u \) is simple, then the cycle is *simple*.

A cycle of length 1 is a self-loop.

A digraph with no self-loops is *simple*.

In an undirected graph, a \( u \)-\( v \) path is a cycle only if simple.

A graph which contains no cycles is *acyclic*.

An acyclic digraph is called a *directed acyclic graph* or *DAG*. 
A Cyclic Digraph

\[ \text{v1} \rightarrow \text{v2} \rightarrow \text{v3} \rightarrow \text{v4} \rightarrow \text{v5} \]
A digraph $G$ is **strongly connected** if for every two distinct vertices $u$ and $v$, there exists a $u$-$v$ path and a $v$-$u$ path.

A graph is **bipartite** if there exists a partition of $V$ into two subsets $V_1$ and $V_2$ such that every edge of $G$ joins a vertex of $V_1$ with $V_2$.

A labeled graph is a triple $\langle V, R, L \rangle$ in which $L$ is a labeling function associated either to the set of vertices or edges.
A Strongly Connected Digraph

\[ \text{v1} \rightarrow \text{v2} \rightarrow \text{v3} \rightarrow \text{v4} \rightarrow \text{v5} \]

\[ \text{v5} \rightarrow \text{v1} \]

Chris J. Myers (Lecture 4: Graphs)
A Simple Labeled Directed Graph

- v1
- v2
- v3
- v4
- v5

Labeled edges:
- a
- b
- c
- d
- e
A Synchronous FSM

INPUTS

Comb. Logic

Register

OUTPUTS

STATE

CLOCK
Finite State Machines

- $I$ is the input alphabet;
- $O$ is the output alphabet;
- $S$ is the finite, non-empty set of states;
- $S_0 \subseteq S$ is the set of initial (reset) states;
- $\delta : S \times I \rightarrow S$ is the next-state function;
- $\lambda : S \times I \rightarrow O$ is the output function for a Mealy machine (or $\lambda : S \rightarrow O$ for a Moore machine).
Finite State Machine Diagrams

- FSM’s are often represented using a labeled digraph.
- The vertex set contains the states (i.e., \( V = S \)).
- The edge set contains the set of state transitions (i.e., \((u, v) \in E \text{ iff } \exists i \in I \text{ s.t. } ((u, i), v) \in \delta\)).
- The labeling function is defined by next-state and output functions.
  - Each edge \((u, v)\) is labeled with \(i/o\) where \(i \in I\) and \(o \in O\) and \(((u, i), v) \in \delta\) and \(((u, i), o) \in \lambda\).
shop_PA_1: process
begin
  guard(req_wine,'1');  -- winery calls
  assign(ack_wine,'1',1,3);  -- receives wine
  guard(req_wine,'0');  -- req_wine reset
  assign(req_patron,'1',1,3);  -- call patron
  guard(ack_patron,'1');  -- wine purchased
  assign(ack_patron,'0',1,3);  -- reset req_patron
  guard(ack_patron,'0');  -- ack_patron reset
  assign(ack_wine,'0',1,3);  -- reset ack_wine
end process;
Passive/Active Shop FSM

\[ \text{start} \]

\[ s0 \]

\[ 10/10 \]

\[ s1 \]

\[ 00/11 \]

\[ 00/00 \]

\[ s2 \]

\[ 01/10 \]

\[ s3 \]

\[ req\_wine / ack\_patron \]

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>s0</td>
<td>00</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>s1</td>
<td>s2</td>
<td>11</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>s2</td>
<td>s2</td>
<td>11</td>
<td>s3</td>
<td>10</td>
</tr>
<tr>
<td>s3</td>
<td>s0</td>
<td>00</td>
<td>s3</td>
<td>10</td>
</tr>
</tbody>
</table>

\[ ack\_wine / req\_patron \]

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>s0</td>
<td>00</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>s1</td>
<td>s2</td>
<td>11</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>s2</td>
<td>s2</td>
<td>11</td>
<td>s3</td>
<td>10</td>
</tr>
<tr>
<td>s3</td>
<td>s0</td>
<td>00</td>
<td>s3</td>
<td>10</td>
</tr>
</tbody>
</table>
Burst-Mode State Machine

\[
\begin{array}{c}
\text{s0} \\
\quad \text{req\_wine+} / \text{ack\_wine+} \\
\quad \text{req\_wine-} / \text{req\_patron+}
\end{array}
\]

\[
\begin{array}{c}
\text{s1} \\
\quad \text{ack\_patron-} / \text{ack\_wine-}
\end{array}
\]

\[
\begin{array}{c}
\text{s2} \\
\quad \text{ack\_patron+} / \text{req\_patron-}
\end{array}
\]

\[
\begin{array}{c}
\text{s3} \\
\text{s0, 00} \quad \text{s3, 10}
\end{array}
\]

\[
\begin{array}{c}
\text{req\_wine / ack\_patron} \\
\text{00} \quad 01 \quad 11 \quad 10
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
\text{s0} & \text{s1, 10} \\
\text{s1} & \text{s2, 11} \quad \text{s1} \quad 10 \\
\text{s2} & \text{s1} \quad 10 \quad \text{s3, 10} \quad \text{--} \\
\text{s3} & \text{s0, 00} \quad \text{s3, 10} \quad \text{--} \quad \text{--}
\end{array}
\]

\[
\begin{array}{c}
\text{ack\_wine / req\_patron}
\end{array}
\]
• $V$ is a finite set of vertices (or states);
• $E \subseteq V \times V$ is the set of edges (or transitions);
• $I = \{x_1, \ldots, x_m\}$ is the set of inputs;
• $O = \{z_1, \ldots, z_n\}$ is the set of outputs;
• $v_0 \in V$ is the start state;
• $\text{in} : V \rightarrow \{0, 1\}^m$ is value of the $m$ inputs at entry to state;
• $\text{out} : V \rightarrow \{0, 1\}^n$ is value of the $n$ outputs at entry to state.
\begin{itemize}
  \item \textit{Input burst} is defined by $\text{trans}_i : E \rightarrow 2^I$.
    \begin{itemize}
      \item $x_i \in \text{trans}_i(e) \text{ iff } \text{in}_i(u) \neq \text{in}_i(v)$
    \end{itemize}
  \item \textit{Output burst} is defined by $\text{trans}_o : E \rightarrow 2^O$.
    \begin{itemize}
      \item $x_i \in \text{trans}_o(e) \text{ iff } \text{out}_i(u) \neq \text{out}_i(v)$
    \end{itemize}
\end{itemize}
Maximal Set Property

- No input burst leaving a given state can be a subset of another leaving the same state.
- The behavior in such a state would be ambiguous.
- $\forall (u, v), (u, w) \in E : trans_i(u, v) \subseteq trans_i(u, w) \Rightarrow v = w$.
- This restriction is called the *maximal set property*. 
Maximal Set Property

\[ s_0 \]

\[ \text{a+} / \text{x+} \]

\[ \text{a+,b+} / \text{y+} \]

\[ s_1 \]

\[ s_2 \]
Not every *BM state diagram* represents a legal BM machine.

If mislabeled with transitions that are not possible, it is impossible to define the *in* and *out* functions.

There must be a strict alternation of rising and falling transitions on every input and output signal, across all paths.
BM State Diagrams

Asynchronous Circuit Design
BM machines require prescribed order: inputs change, outputs change, and state signals change.

In extended burst-mode (XBM) state machines, this limitation is loosened a bit by the introduction of directed don’t cares.

These allow one to specify that an input change may or may not happen in a given input burst.

BM machines also are unable to express conditional behavior.

To support this type of behavior, XBM machines allow conditional input bursts.
Directed Don’t Cares

\[ \text{Shop\_PA\_2 : process} \]
\[ \text{begin} \]
  \[ \text{guard(req\_wine, '1'); \quad \text{-- winery calls} \]  
  \[ \text{assign(ack\_wine, '1', 1, 3); \quad \text{-- receives wine} \]  
  \[ \text{guard(req\_wine, '0'); \quad \text{-- req\_wine reset} \]  
  \[ \text{assign(ack\_wine = '0', 1, 3, req\_patron, '1', 1, 3); \]  
  \[ \text{guard(ack\_patron, '1'); \quad \text{-- wine purchased} \]  
  \[ \text{assign(req\_patron, '0', 1, 3); \quad \text{-- reset req\_patron} \]  
  \[ \text{guard(ack\_patron, '0'); \quad \text{-- ack\_patron reset} \]  
\[ \text{end process;} \]
Directed Don’t Cares

s0

req_wine+/ ack_wine+

s1

req_wine-/ ack_wine-, req_patron+

s2

req_wine*, ack_patron+/
req_patron-

s3

req_wine+, ack_patron-/ ack_wine+
Directed Don’t Cares

### req_wine / ack_patron

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>s0</td>
<td>00</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>s1</td>
<td>s2,</td>
<td>01</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>s2</td>
<td>s2</td>
<td>01</td>
<td>s3,</td>
<td>00</td>
</tr>
<tr>
<td>s3</td>
<td>s3</td>
<td>00</td>
<td>s3</td>
<td>00</td>
</tr>
</tbody>
</table>

### ack_wine / req_patron
Directed Don’t Cares

- A transition is *terminating* when it is of the form $t+$ or $t−$.
- A directed don’t care transition is of the form $t*$.
- A *compulsory* transition is a terminating transition which is not preceded by a directed don’t care transition.
- Each input burst must have at least one compulsory transition.
Directed Don’t Cares

- $s_0$: req_wine+/ack_wine+
- $s_1$: req_wine-/ack_wine-, req_patron+
- $s_2$: req_wine*, ack_patron+/req_patron-
- $s_3$: req_wine*, ack_patron-/
Modified Maximal Set Property

\[ a^+, b^*/ x^+ \]
\[ s_0 \]
\[ \rightarrow \]
\[ s_1 \]
\[ b^+/ y^+ \]
\[ \rightarrow \]
\[ s_2 \]

\[ a^+, b^*/ x^+ \]
\[ \rightarrow \]
\[ s_1 \]

\[ a^+, b^+\]
Shop_PA_2: process  
begin  
guard(req_wine,'1');  
shelf <= bottle after delay(2,4);  
wait for delay(5,10);  
assign(ack_wine,'1',1,3);  
guard(req_wine,'0');  
if (shelf = '0') then  
  assign(ack_wine,'0',1,3,req_patron1,'1',1,3);  
guard(ack_patron1,'1');  
assign(req_patron1,'0',1,3);  
guard(ack_patron1,'0');
elsif (shelf = '1') then
    assign(ack_wine,'0',1,3,req_patron2,'1',1,3);
    guard(ack_patron2,'1');
    assign(req_patron2,'0',1,3);
    guard(ack_patron2,'0');
end if;
end process;
A conditional input burst includes a regular input burst and a *conditional clause*.

A clause of the form $< s \neg >$ indicates that the transition is only taken if $s$ is low.

A clause of the form $< s \neg >$ indicates that the transition is only taken if $s$ is high.

The signal in the conditional clause must be stable before every compulsory transition in the input burst.
Conditional Input Bursts

Asynchronous Circuit Design
## Conditional Input Bursts

<table>
<thead>
<tr>
<th></th>
<th>req_wine</th>
<th>ack_patron1</th>
<th>ack_patron2</th>
<th>shelf</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>0000</td>
<td>0000</td>
<td>0011</td>
<td>0010</td>
</tr>
<tr>
<td>s1</td>
<td>s2, 010</td>
<td>s4, 001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>s2, 010</td>
<td>s2, 010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>s3, 010</td>
<td>s3, 010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>s4, 001</td>
<td>s4, 001</td>
<td>s5, 000</td>
<td>s5, 000</td>
</tr>
<tr>
<td>s5</td>
<td>s5, 000</td>
<td>s5, 000</td>
<td>s5, 000</td>
<td>s5, 000</td>
</tr>
</tbody>
</table>

## Ack_Wine / Req_Patron1 / Req_Patron2
## Conditional Input Bursts

<table>
<thead>
<tr>
<th></th>
<th>req_wine / ack_patron1 / ack_patron2 / shelf</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1100</td>
</tr>
<tr>
<td>s0</td>
<td>–</td>
</tr>
<tr>
<td>s1</td>
<td>–</td>
</tr>
<tr>
<td>s2</td>
<td>s3, 000</td>
</tr>
<tr>
<td>s3</td>
<td>s3, 000</td>
</tr>
<tr>
<td>s4</td>
<td>–</td>
</tr>
<tr>
<td>s5</td>
<td>–</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ack_wine / req_patron1 / req_patron2</th>
</tr>
</thead>
<tbody>
<tr>
<td>s5</td>
<td>s1, 100</td>
</tr>
</tbody>
</table>
Modified Maximal Set Property

\[ s_0 \]

\[ s_1 \rightarrow <s+>a+,b+/ x+ \]

\[ s_2 \rightarrow <s->a+/ y+ \]

\[ s_0 \]

\[ s_1 \rightarrow <s+>a+,b*/ x+ \]

\[ s_2 \rightarrow <s->a*,b+/ y+ \]

\[ s_0 \]

\[ s_1 \rightarrow <s+>a+,b+/ x+ \]

\[ s_2 \rightarrow <s->a+,b+/ y+ \]
Burst-Mode State Machines

- $V$ is a finite set of vertices (or states).
- $E \subseteq V \times V$ is the set of edges (or transitions).
- $I = \{x_1, \ldots, x_m\}$ is the set of inputs.
- $O = \{z_1, \ldots, z_n\}$ is the set of outputs.
- $C = \{c_1, \ldots, c_l\}$ is the set of conditional signals.
- $v_0 \in V$ is the start state.
- $in : V \rightarrow \{0, 1, *\}^m$ defines $m$ inputs upon entry to each state.
- $out : V \rightarrow \{0, 1\}^n$ defines $n$ outputs upon entry to each state.
- $cond : E \rightarrow \{0, 1, *\}^l$ defines needed conditional inputs.
Shop_PA_lazy_active: process
begin
  guard(req_wine,'1');       - winery calls
  assign(ack_wine,'1',1,3);  - receives wine
  guard(ack_patron,'0');     - ack_patron reset
  assign(req_patron,'1',1,3); - call patron
  guard(req_wine,'0');       - req_wine reset
  assign(ack_wine,'0',1,3);  - reset ack_wine
  guard(ack_patron,'1');     - wine purchased
  assign(req_patron,'0',1,3); - reset req_patron
end process;
Illegal XBM Machine

s0

 req_wine+,ack_patron*/
  ack_wine+

s1

 req_wine*,ack_patron-/  
  req_patron+

s2

 req_wine-,ack_patron*/  
  ack_wine-

s3

 req_wine*,ack_patron+/  
  req_patron-
A Petri-net is a bipartite digraph.

The vertex set is partitioned into two disjoint subsets:
- $P$ is the set of places.
- $T$ is the set of transitions.

The set of arcs, $F$, is composed of pairs where one element is from $P$ and the other is from $T$ (i.e., $F \subseteq (P \times T) \cup (T \times P)$).

A Petri-net is $\langle P, T, F, M_0 \rangle$ where $M_0$ is the initial marking.
Petri-net for Shop with Infinite Shelf Space

Asynchronous Circuit Design

Chris J. Myers (Lecture 4: Graphs)
The *preset* of a transition \( t \in T \) (denoted \( \bullet t \)) is the set of places connected to \( t \) (i.e., \( \bullet t = \{ p \in P \mid (p, t) \in F \} \)).

The *postset* of a transition \( t \in T \) (denoted \( t \bullet \)) is the set of places \( t \) is connected to (i.e., \( t \bullet = \{ p \in P \mid (t, p) \in F \} \)).

The preset of a place \( p \in P \) (denoted \( \bullet p \)) is the set of transitions connected to \( p \) (i.e., \( \bullet p = \{ t \in T \mid (t, p) \in F \} \)).

The postset of a place \( p \in P \) (denoted \( p \bullet \)) is the set of transitions \( p \) is connected to (i.e., \( p \bullet = \{ t \in T \mid (p, t) \in F \} \)).
A marking, $M$, for a Petri net is a function that maps places to natural numbers (i.e., $M : P \rightarrow N$).

Markings can be added or subtracted using vector arithmetic.

They can also be compared:

$$M \geq M' \iff \forall p \in P . \ M(p) \geq M'(p)$$

For a set of places, $A \subseteq P$, $C_A$ denotes the characteristic marking of $A$:

$$C_A(p) = \text{if } p \in A \text{ then } 1 \text{ else } 0.$$
A transition $t$ is *enabled* under the marking $M$ if $M \geq C_{\bullet t}$.

In other words, $M(p) \geq 1$ for each $p \in \bullet t$.

The firing transforms the marking as follows (denoted $M[t]M'$):

$$M' = M - C_{\bullet t} + C_{t\bullet}$$

When a transition $t$ fires, a token is removed from each place in its preset, and a token is added to each place in its postset.
Firing of a transition transforms the marking of the Petri net into a new marking.

A sequence of transition firings \((\sigma = t_1, t_2, \ldots, t_n)\) produces a sequence of markings \((M_0, M_1, \ldots, M_n)\).

If such a firing sequence exists, we say that the marking \(M_n\) is reachable from \(M_0\) by \(\sigma\) (denoted \(M_0[\sigma]M_n\)).

We denote the set of all markings reachable from a given marking by \([M]\).
Example Firing Sequence

Asynchronous Circuit Design
Example Firing Sequence

1. **produce**
2. **receive**
3. **send**
4. **consume**
Example Firing Sequence

Asynchronous Circuit Design

1. produce
2. receive
3. send
4. consume

Diagram:
- Produce (1)
- Receive (2)
- Send (3)
- Consume (4)
Example Firing Sequence

produce → receive → 2 → send
1

consume
1
1

1
Example Firing Sequence

produce

receive

send

consume

1

1

2
Example Firing Sequence

produce \rightarrow receive \rightarrow send \rightarrow consume

1 \rightarrow 1 \rightarrow 1

Asynchronous Circuit Design
Example Firing Sequence

produce → receive → send → consume

1

2
A Petri net is *k*-bounded if there does not exist a reachable marking which has a place with more than *k* tokens.

A 1-bounded Petri net is also called a safe Petri net (i.e., \(\forall p \in P, \forall M \in [M_0]. M(p) \leq 1\)).

When working with safe Petri nets, a marking can be denoted as simply a subset of places.

If \(M(p) = 1\), \(p \in M\), and if \(M(p) = 0\), we \(p \notin M\).

\(M(p)\) cannot take on any other values in a safe Petri net.

Since a marking can only take on the values 1 and 0, the place can be annotated with a token when 1 and without when 0.
$k$-Bounded Petri-net

![Petri-net Diagram]

- **Produce**: 1
- **Receive**: 2
- **Send**: 1
- **Consume**: 2

Chris J. Myers (Lecture 4: Graphs)
A Petri net is *live* if from every reachable marking, there exists a sequence of transitions such that any transition can fire.

\[
\forall M \in [M_0], \forall t \in T, \exists M' \in [M]. M' \geq C_t
\]

To determine if a Petri net is live, it is typically necessary to find all the reachable markings.
Liveness

produce ← receive → send → consume

Chris J. Myers (Lecture 4: Graphs)  Asynchronous Circuit Design
Different liveness categories can be determined more easily.

In particular, a transition $t$ for a given Petri net is said to be:

1. **dead (L0-live)** if there does not exist a firing sequence in which $t$ can be fired.
2. **L1-live (potentially firable)** if there exists at least one firing sequence in which $t$ can be fired.
3. **L2-live** if $t$ can be fired at least $k$ times.
4. **L3-live** if $t$ can be fired infinitely often in some firing sequence.
5. **L4-live or live** if $t$ is L1-live in every marking reachable from the initial marking.

A Petri net is **Lk-live** if every transition in the net is Lk-live.
Liveness Categories

produce 1 deliver consume receive
When a Petri net is bounded, the number of reachable markings is finite, and a \textit{reachability graph} (RG) can be found.

In an RG, the vertices, $\Phi$, are the markings and the edges, $\Gamma$, are the possible transition firings between two markings.

For safe Petri nets, vertices in RG are labeled with the subset of places included in the marking.

The edges are labeled with the transition that fires to move the Petri net from one marking to the next.
Algorithm to Find Reachability Graph

\textbf{Algorithm  A}{\text{RG}}(\text{Petri net } \langle P, T, F, M_0 \rangle)

\begin{align*}
M &= M_0; \\
T_e &= \{ t \in T | M \geq C \cdot t \}; \\
\Phi &= \{ M \}; \\
\Gamma &= \emptyset;
\end{align*}

done = false;

\textbf{while} (¬ done)

\begin{align*}
t &= \text{select}(T_e);
\text{if } (T_e - \{ t \} \neq \emptyset) \text{ then push}(M, T_e - \{ t \});
M' &= M - C \cdot t + C_t \cdot ;
\text{if } (M' \notin \Phi) \text{ then }
\Phi &= \Phi \cup \{ M' \}; \\
\Gamma &= \Gamma \cup \{ (M, M') \} ; \\
M &= M'; \\
T_e &= \{ t \in T | M \geq C \cdot t \};
\text{else }
\Gamma &= \Gamma \cup \{ (M, M') \} ;
\text{if (stack is not empty) then } (M, T_e) = \text{pop}();
\text{else } \text{done } = \text{true};
\text{return}(\Phi, \Gamma);
\end{align*}
Safe Example

Asynchronous Circuit Design

Chris J. Myers  (Lecture 4: Graphs)  Asynchronous Circuit Design  59 / 105
Example Reachability Graph

Chris J. Myers  (Lecture 4: Graphs)  Asynchronous Circuit Design
Two transitions $t_1$ and $t_2$ are *concurrent* when there exists markings where both are enabled and can fire in either order.

Two transitions, $t_1$ and $t_2$, are in *conflict* when the firing of one disables the firing of the other.

When concurrency and conflict are mixed, we get *confusion*. 
Example of Concurrency, Conflict, and Confusion
A Petri-net is a *state machine* if and only if every transition has exactly one place in its preset and one place in its postset.

\[ \forall t \in T : |\bullet t| = |t \bullet| = 1 \]

State machines do not allow concurrency, but do allow conflict.

A Petri-net is a *marked graph* if and only if every place has exactly one transition in its preset and one in its postset.

\[ \forall p \in P : |\bullet p| = |p \bullet| = 1 \]

Marked graphs do not allow conflict, but do allow concurrency.
Example Nets

Asynchronous Circuit Design
A Petri-net is *free choice* if and only if every pair of transitions that share a common place in their preset have only a single place in their preset.

\[ \forall t, t' \in T, t \neq t' : t \cap t' \neq \emptyset \Rightarrow |t| = |t'| = 1 \]

\[ \forall p, p' \in P, p \neq p' : p \cap p' \neq \emptyset \Rightarrow |p| = |p'| = 1 \]

\[ \forall p \in P, \forall t \in T : (p, t) \in F \Rightarrow p \cap t = \{p\} \]

Free choice nets allow concurrency and conflict, but do allow confusion.
Example Nets

[Diagram of example nets with nodes labeled as: produce, receive1, receive2, send1, send2, consume1, consume2]
A Petri net is an *extended free choice net* if and only if every pair of places that share common transitions in their postset have exactly the same transitions in their postset.

\[ \forall p, p' \in P . \ p \bullet \cap p' \bullet \neq \emptyset \Rightarrow p \bullet = p' \bullet \]

- Extended free-choice nets also allow concurrency and conflict, but they do not allow confusion.
Example Nets

Asynchronous Circuit Design

Chris J. Myers  (Lecture 4: Graphs)
A Petri net is an *asymmetric choice net* if and only if for every pair of places that share common transitions in their postset, one has a subset of the transitions of the other.

\[ \forall p, p' \in P . \ p \bullet \cap p' \bullet \neq \emptyset \Rightarrow p \bullet \subseteq p' \bullet \lor p' \bullet \subseteq p \bullet \]

- Asymmetric choice nets allow *asymmetric confusion* but not *symmetric confusion*. 
Example Nets

Asynchronous Circuit Design
It is possible to check safety and liveness for certain restricted classes of Petri nets using the theorems given below.

**Theorem 4.1** A state machine is live and safe iff it is strongly connected and $M_0$ has exactly one token.

**Theorem 4.2 (Commoner, 1971)** A marked graph is live and safe iff it is strongly connected and $M_0$ places exactly one token on each simple cycle.
A *siphon* is a nonempty subset of places, $S$, in which every transition having a postset place in $S$ also has a preset place in $S$ (i.e., $\bullet S \subseteq S\bullet$).

If in some marking no place in $S$ has a token, then in all future markings, no place in $S$ will ever have a token.

A *trap* is a nonempty subset of places, $Q$, in which every transition having a preset place in $Q$ also has a postset place in $Q$ (i.e., $Q\bullet \subseteq \bullet Q$).

If in some marking some place in $Q$ has a token, then in all future markings some place in $Q$ will have a token.
Example Siphon and Trap

S

\[ \text{S} \]

Q

\[ \text{Q} \]
Theorem 4.3 (Hack, 1972) A free-choice net, $N$, is live iff every siphon in $N$ contains a marked trap.

Theorem 4.4 (Commoner, 1972) An asymmetric choice net $N$ is live if (but not only if) every siphon in $N$ contains a marked trap.
State Machine Components

- A *state machine component* of a net, $N$, is a subnet in which each transition has at most one place in its preset and one place in its postset and is generated by these places.
- The net generated by a set of places includes these places, all transitions in their preset and postset, and all connecting arcs.
- A net $N$ is said to be covered by a set of SM-components when the set of components includes all places, transitions, and arcs from $N$.

**Theorem 4.5 (Hack, 1972)** A live free-choice net, $N$, is safe iff $N$ is covered by strongly connected SM-components each of which has exactly one token in $M_0$. 

Chris J. Myers  (Lecture 4: Graphs)  Asynchronous Circuit Design  76 / 105
A marked graph component of a net, $N$, is a subnet in which each place has at most one transition in its preset and one transition in its postset and is generated by these transitions.

The net generated by a set of transitions includes these transitions, all places in their preset and postset, and all connecting arcs.

A net $N$ is said to be covered by a set of MG-components when the set of components includes all places, transitions, and arcs from $N$.

**Theorem 4.6** If $N$ is a live and safe free-choice net then $N$ is covered by strongly connected MG-components.
To use a Petri net to model asynchronous circuits, must relate transitions to events on signal wires.

Several variants of Petri nets accomplish this: *M-nets*, *I-nets*, and *change diagrams*.

A *signal transition graph* (STG) is a labeled safe Petri net which is modeled by \( \langle P, T, F, M_0, N, s_0, \lambda_T \rangle \), where:

- \( N = I \cup O \) is the set of signals where \( I \) is the set of input signals and \( O \) is the set of output signals.
- \( s_0 \) is the initial value for each signal in the initial state.
- \( \lambda_T : T \rightarrow N \times \{+, -\} \) is the *transition labeling function*.

Each transition is labeled with either a rising transition, \( s+ \), or falling transition, \( s- \).

A STG imposes explicit restrictions on the environment.
Example Signal Transition Graph (STG)
STGs are often restricted to a synthesizable subset.

- Synthesis methods often restrict the STG to be live and safe.
- Some synthesis methods require STGs to be persistent.

A STG is persistent if for all \( a^* \rightarrow b^* \), there exist other arcs that ensure that \( b^* \) fires before the opposite transition of \( a^* \).

- Other methods require single-cycle transitions.

A STG has single-cycle transitions if each signal name appears in exactly one rising and one falling transition.

- None of these restrictions is actually a necessary requirement for a circuit implementation to exist.
- These restrictions can simplify the synthesis algorithms.
Liveness

reset

r+

x+

y-

x-

y+
Safety
Persistency
Single-Cycle Transitions

Chris J. Myers (Lecture 4: Graphs)
To design a circuit from an STG, must find its state graph.

A SG is modeled by the tuple \( \langle S, \delta, \lambda_S \rangle \).

- \( S \) is the set of states.
- \( \delta \subseteq S \times T \times S \) is the set of state transitions.
- \( \lambda_S : S \rightarrow (N \rightarrow \{0, 1\}) \) is the state labeling function.

Each state \( s \) is labeled with a vector \( \langle s(0), s(1), \ldots, s(n) \rangle \), where \( s(i) \) is either 0 or 1, indicating value returned by \( \lambda_S \).

We use \( s(i) \) interchangeably with \( \lambda_S(s)(i) \).
Implied State

- If in $s_i$, there exists a transition on signal $u_i$ to $s_j$ [i.e.,
  $\exists (s_i, t, s_j) \in \delta . \lambda_T(t) = u_i + \lor \lambda_T(t) = u_i-$], then $u_i$ is excited.
- Otherwise, the signal $u_i$ is in equilibrium.
- The value each signal is tending to is called its implied value.
- If the signal is excited, the implied value of $u_i$ is $s(i)$.
- If the signal is in equilibrium, the implied value of $u_i$ is $s(i)$.
- The implied state, $s'$ is labeled with a binary vector $\langle s'(0), s'(1), \ldots, s'(n) \rangle$ of the implied values.
- The function $X : S \rightarrow 2^N$ returns the set of excited signals in a given state
  [i.e., $X(s) = \{ u_i \in S \mid s(i) \neq s'(i) \}$].
- When $u_i \in X(s)$ and $s(i) = 0$, $s(i)$ in SG is marked with “R”.
- When $u_i \in X(s)$ and $s(i) = 1$, $s(i)$ in SG is marked with “F”.

Chris J. Myers (Lecture 4: Graphs)
Algorithm to Find SG

\[\text{find\_SG}(\langle P, T, F, M_0, N, s_0, \lambda_T \rangle)\]

\[M = M_0; \ s = s_0; \ S = \{M\}; \ \lambda_S(M) = s;\]

\[T_e = \{t \in T | M \subseteq \bullet t\}; \ \text{done} = \text{false};\]

\textbf{while} (\neg \text{done})

\[t = \text{select}(T_e);\]

\textbf{if} \ (T_e - \{t\} \neq \emptyset) \ \textbf{then} \ \text{push}(M, s, T_e - \{t\});

\textbf{if} \ ((M - \bullet t) \cap \bullet t \neq \emptyset) \ \textbf{then} \ \text{return}(\text{"Not safe."});

\[M' = (M - \bullet t) \cup \bullet t; \ s' = s;\]

\textbf{if} \ (\lambda_T(t) = u++) \ \textbf{then} \ s'(u) = 1;

\textbf{else if} \ (\lambda_T(t) = u--) \ \textbf{then} \ s'(u) = 0;

\textbf{if} \ (M' \not\in S) \ \textbf{then}

\[S = S \cup \{M'\}; \ \lambda_S(M') = s' \ \delta = \delta \cup \{(M, t, M')\};\]

\[M = M'; \ s = s'; \ T_e = \{t \in T | M \subseteq \bullet t\};\]

\textbf{else}

\textbf{if} \ (\lambda_S(M') \neq s') \ \textbf{then} \ \text{return}(\text{"Inconsistent."});

\textbf{if} \ (\text{stack is not empty}) \ \textbf{then} \ (M, s, T_e) = \text{pop}();

\textbf{else} \ \text{done} = \text{true};

\textbf{return}(\langle S, \delta, \lambda_S \rangle);
Example Signal Transition Graph (STG)
A well-formed SG, must have a *consistent state assignment*. A SG has a consistent state assignment if for each state transition $(s_i, t, s_j) \in \delta$ exactly one signal changes value, and its value is consistent with the transition.

\[
\forall (s_i, t, s_j) \in \delta. \forall u \in N \quad (\lambda_T(t) \neq u^* \land s_i(u) = s_j(u))
\]

\[
\lor (\lambda_T(t) = u + \land s_i(u) = 0 \land s_j(u) = 1)
\]

\[
\lor (\lambda_T(t) = u - \land s_i(u) = 1 \land s_j(u) = 0)
\]

where “*” represents either “+” or “−”.

A STG produces a SG with a consistent state assignment if in any firing sequence, transitions of a signal strictly alternate between +’s and −’s.
A SG has a *unique state assignment* (USC) if no two different states (i.e., markings) have identical values for all signals [i.e., \( \forall s_i, s_j \in S, s_i \neq s_j \implies \lambda(s_i) \neq \lambda(s_j) \)].

Some synthesis methods are restricted to STGs that produce SGs with USC.
Reshuffled Passive/Lazy-Active Wine Shop

Shop_PA_lazy_active: process
begin
  guard(req_wine,'1');         -- winery calls
  assign(ack_wine,'1',1,3);    -- receives wine
  guard(ack_patron,'0');       -- ack_patron reset
  assign(req_patron,'1',1,3);  -- call patron
  guard(req_wine,'0');         -- req_wine reset
  assign(ack_wine,'0',1,3);    -- reset ack_wine
  guard(ack_patron,'1');       -- wine purchased
  assign(req_patron,'0',1,3);  -- reset req_patron
end process;
Shop_PA_2 : process
begin
    guard(req_wine,'1');
    shelf <= bottle after delay(2,4);
    wait for delay(5,10);
    assign(ack_wine,'1',1,3);
    guard(req_wine,'0');
    if (shelf = '0') then
        assign(ack_wine,'0',1,3,req_patron1,'1',1,3);
        guard(ack_patron1,'1');
        assign(req_patron1,'0',1,3);
        guard(ack_patron1,'0');
elsif (shelf = '1') then
    assign(ack_wine,'0',1,3,req_patron2,'1',1,3);
    guard(ack_patron2,'1');
    assign(req_patron2,'0',1,3);
    guard(ack_patron2,'0');
end if;
end process;
Labeled Petri nets

- AFSMs cannot model arbitrary concurrency.
- Petri-nets have difficulty to express signal levels.
- *Labeled Petri nets* are a hybrid graphical representation method which are both capable of modelling arbitrary concurrency and signal levels.
A LPN is a tuple $\langle P, T, B, F, L, M_0, S_0 \rangle$ where:

- $P$ is a finite set of places;
- $T$ is a finite set of transitions;
- $B$ is a finite set of Boolean variables;
- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation;
- $L$ is a tuple of labels;
- $M_0 \subseteq P$ is the set of initially marked places; and
- $S_0$ is the set of initial Boolean variable values.
Each transition \( t \in T \) has the following labels:

- \( \text{En} : T \rightarrow P \) - the enabling condition;
- \( \text{D} : T \rightarrow \mathbb{Q} \times (\mathbb{Q} \cup \{\infty\}) \) - the delay assignment; and
- \( \text{BA} : T \times B \rightarrow P \) - Boolean variable assignments.

The language for the \( P \) is defined as follows:

\[
\phi ::= \text{true} | \text{false} | b_i | \neg \phi | \phi \land \phi | \phi \lor \phi
\]
A transition $t$ is enabled to fire when its preset ($\bullet t$) is marked and its enabling condition ($En(t)$) evaluates to true in the current state.

Once a transition is enabled it fires sometime between the lower and upper bound associated with its delay assignment ($D(t)$).

When a transition fires, the marking is updated and the Boolean assignments associated with the transition ($BA(t, v)$) are executed.
LPN for a C-Element

\[
\begin{align*}
\text{p0} & \xrightarrow{t0} \text{p1} & & \{\neg z\} \quad <x:=true> \\
\text{p1} & \xrightarrow{t1} \text{p0} & & \{z\} \quad <x:=false> \\
\text{p2} & \xrightarrow{t2} \text{p3} & & \{\neg z\} \quad <y:=true> \\
\text{p3} & \xrightarrow{t3} \text{p2} & & \{z\} \quad <y:=false> \\
\text{p4} & \xrightarrow{t4} \text{p5} & & \{x\land y\} \quad <z:=true> \\
\text{p5} & \xrightarrow{t5} \text{p4} & & \{\neg x\land\neg y\} \quad <z:=false>
\end{align*}
\]
LPN for Wine Shop with Two Patrons
Finite state machines (AFSMs, BM, and XBM).
Petri-nets, STGs, and LPNs.