

# **Asynchronous Circuit Design**

**Chris J. Myers**

**Lecture 4: Graphical Representations**

**Chapter 4**

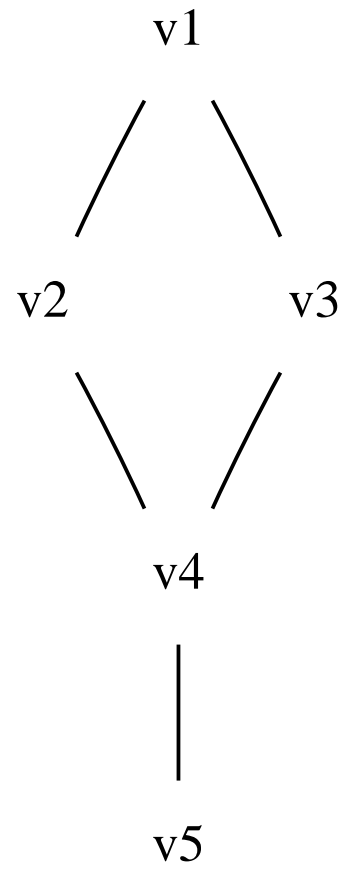
## Chapter Overview

- 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
  - TEL structures

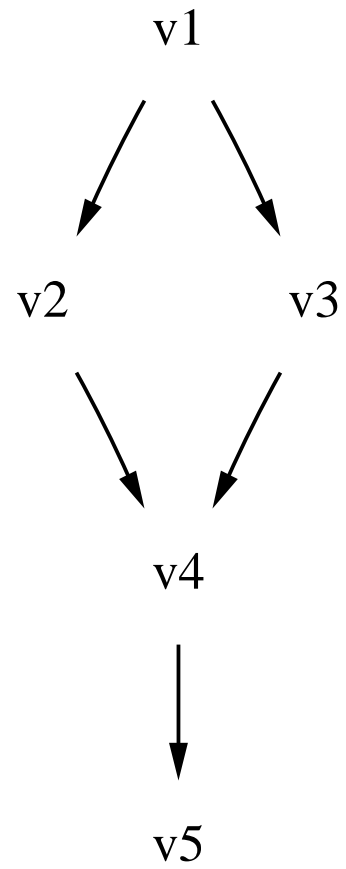
## Graph Basics

- 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



## A Simple Directed Graph



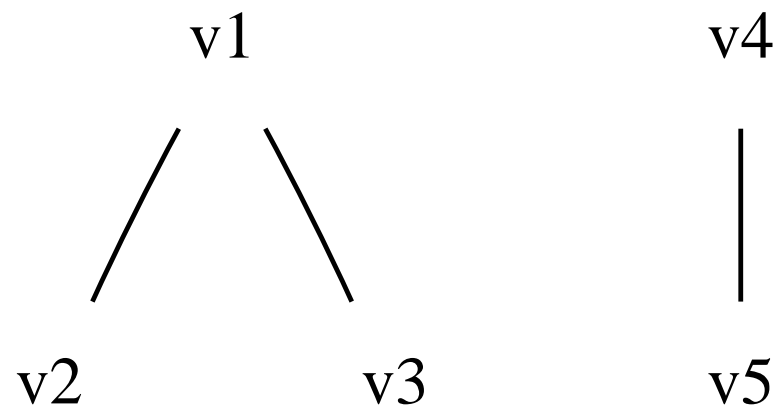
## Additional Graph Definitions

- $|V|$  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  $(u, v) \notin E(G)$ ,  $u$  and  $v$  are *nonadjacent* vertices.

## Connected Graphs

- $u$ - $v$  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.

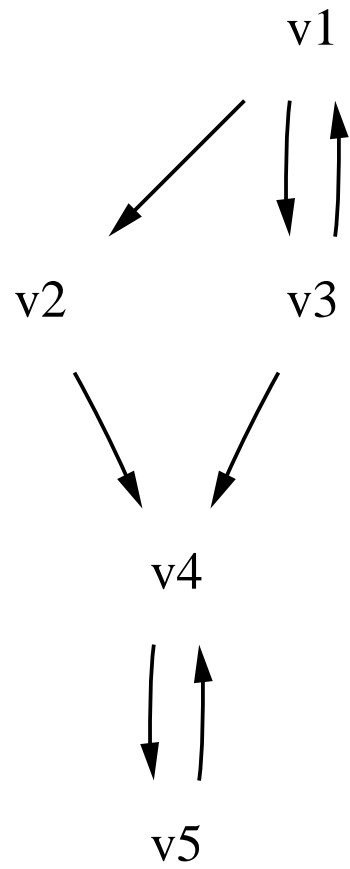
## A Unconnected Graph



## Directed Acyclic Graphs

- 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*.

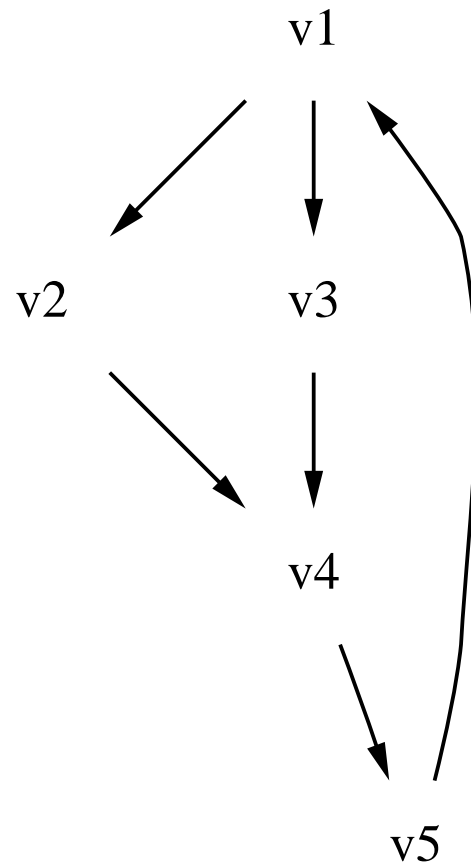
## A Cyclic Digraph



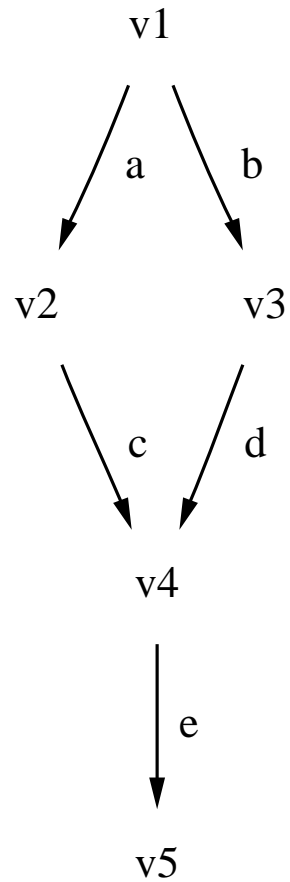
## More Graph Properties

- 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.

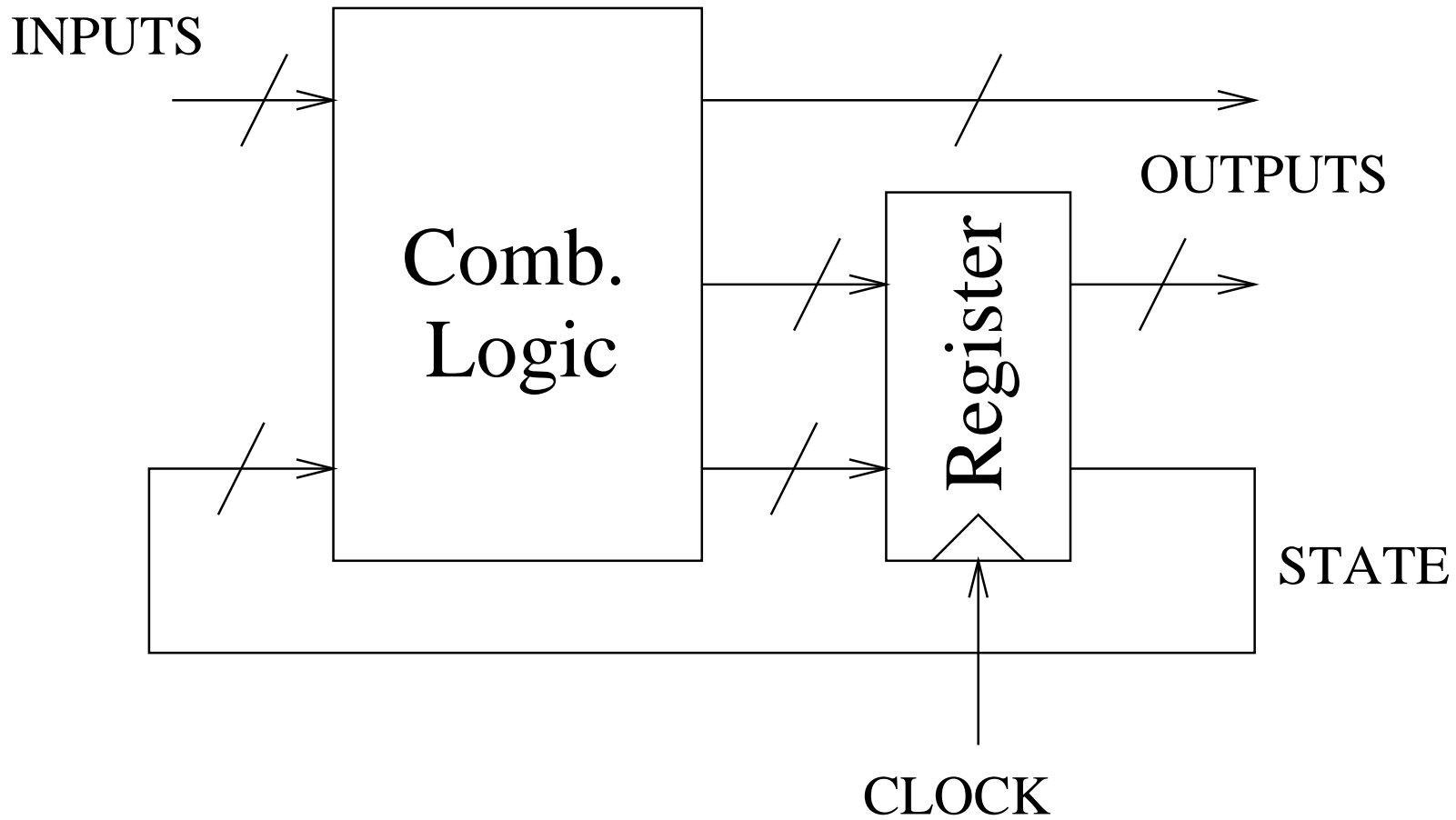
## A Strongly Connected Digraph



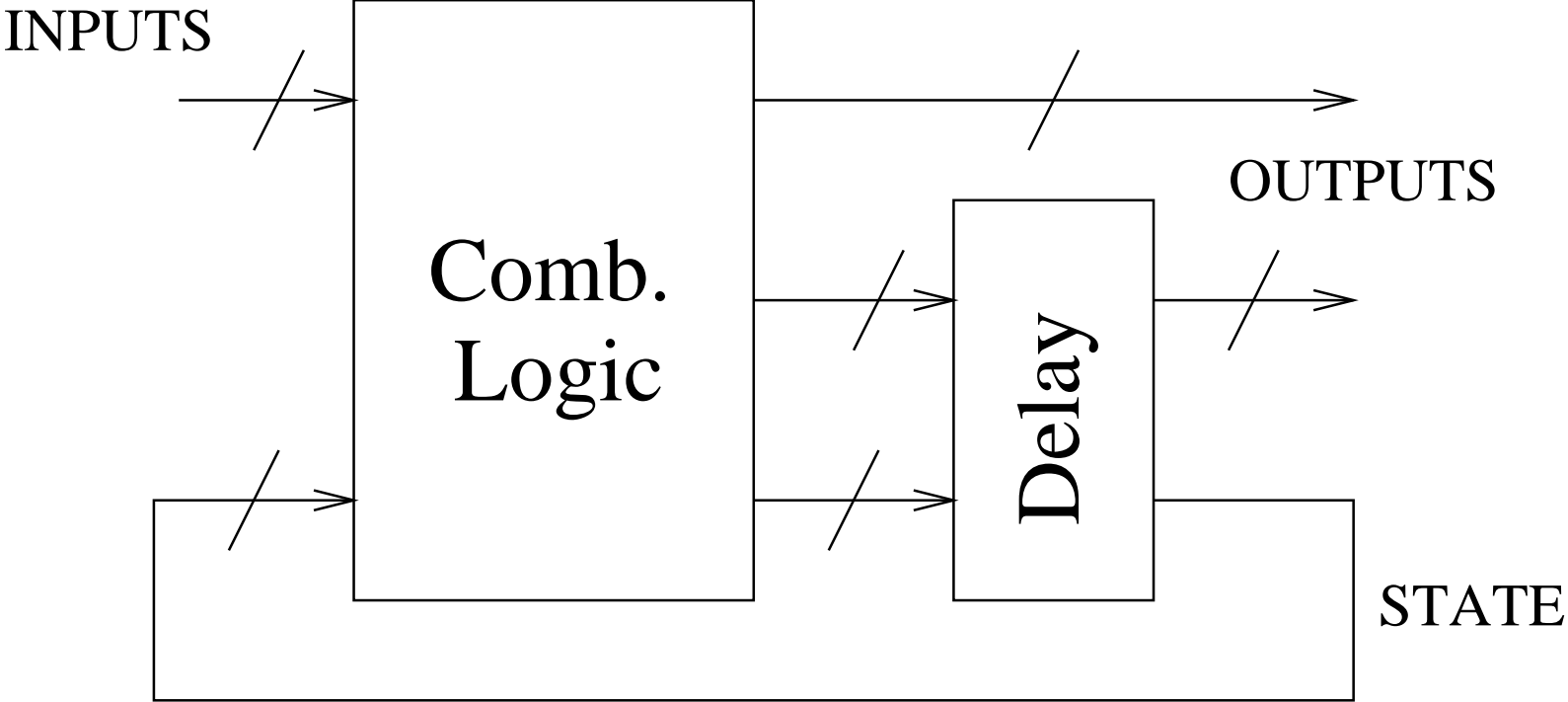
## A Simple Labeled Directed Graph



# A Synchronous FSM



# An Asynchronous FSM



## 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).

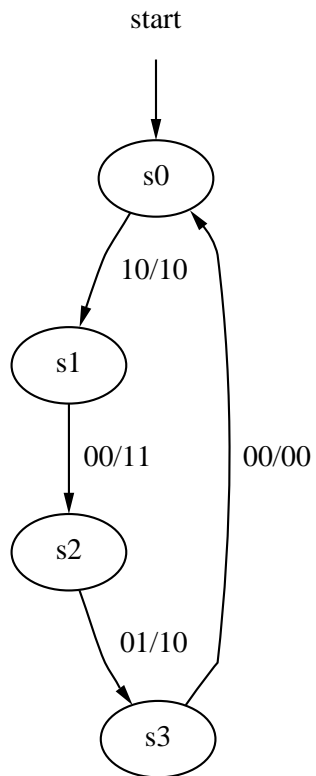
## Finite State Machine Diagrams

- 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.,  $(u, v) \in E$  iff  $\exists i \in I$  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  $((u, i), o) \in \lambda$ .

## Passive/Active Shop

```
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(req_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;
```

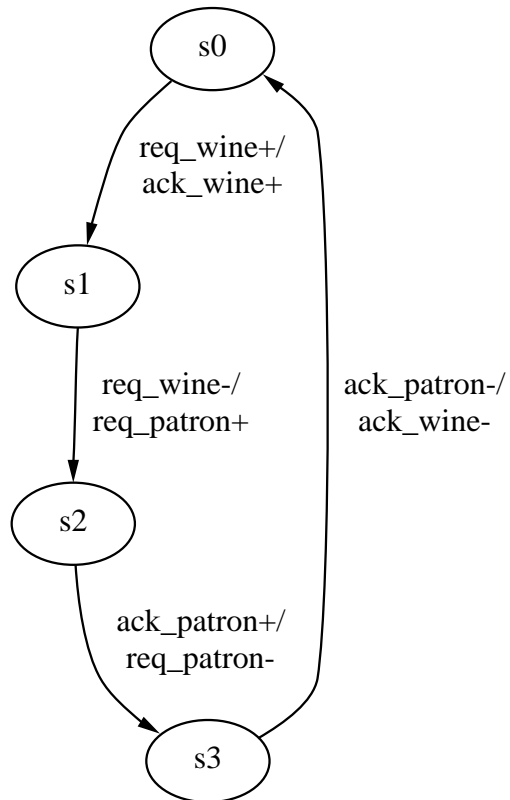
## Passive/Active Shop FSM



	<i>req_wine / ack_patron</i>			
	00	01	11	10
s0	⓪, 00	—	—	s1, 10
s1	s2, 11	—	—	Ⓛ, 10
s2	Ⓜ, 11	s3, 10	—	—
s3	s0, 00	Ⓝ, 10	—	—

*ack\_wine / req\_patron*

## Burst-Mode State Machine



		<i>req_wine / ack_patron</i>			
		00	01	11	10
s0	⓪, 00	—	—	s1, 10	
s1	s2, 11	—	—	Ⓛ, 10	
s2	Ⓜ, 11	s3, 10	—	—	
s3	s0, 00	Ⓨ, 10	—	—	

*ack\_wine / req\_patron*

## 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);
- $I = \{x_1, \dots, x_m\}$  is the set of inputs;
- $O = \{z_1, \dots, z_n\}$  is the set of outputs;
- $v_0 \in V$  is the start state;
- $in : V \rightarrow \{0, 1\}^m$  is value of the  $m$  inputs at entry to state;
- $out : V \rightarrow \{0, 1\}^n$  is value of the  $n$  outputs at entry to state.

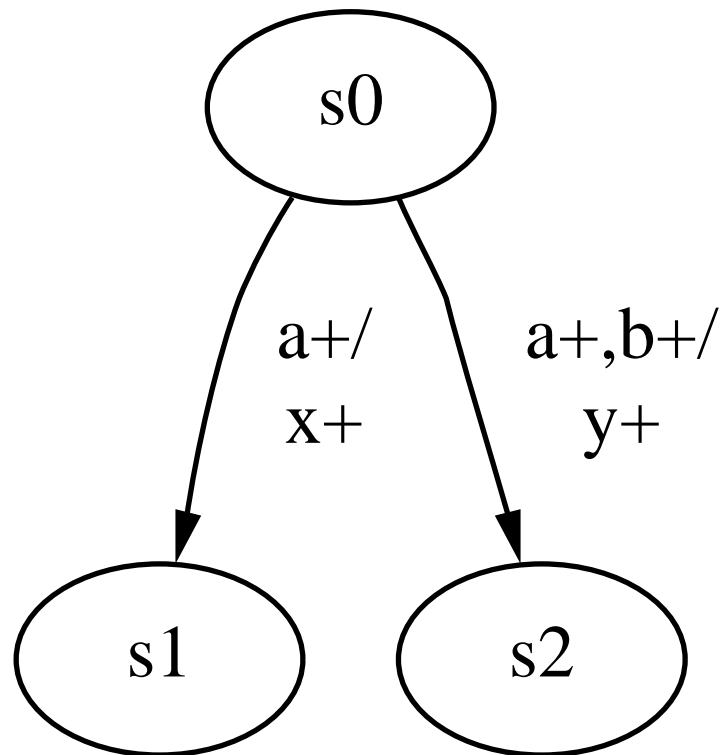
## Input and Output Bursts

- *Input burst* is defined by  $trans_i : E \rightarrow 2^I$ .
  - $x_i \in trans_i(e)$  iff  $in_i(u) \neq in_i(v)$
- *Output burst* is defined by  $trans_o : E \rightarrow 2^O$ .
  - $x_i \in trans_o(e)$  iff  $out_i(u) \neq out_i(v)$

## 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*.

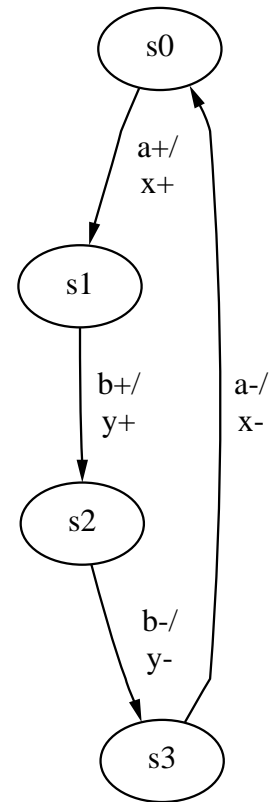
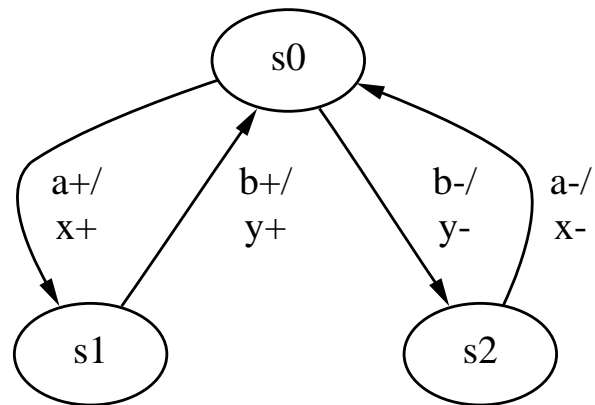
## Maximal Set Property



## BM State Diagrams

- 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



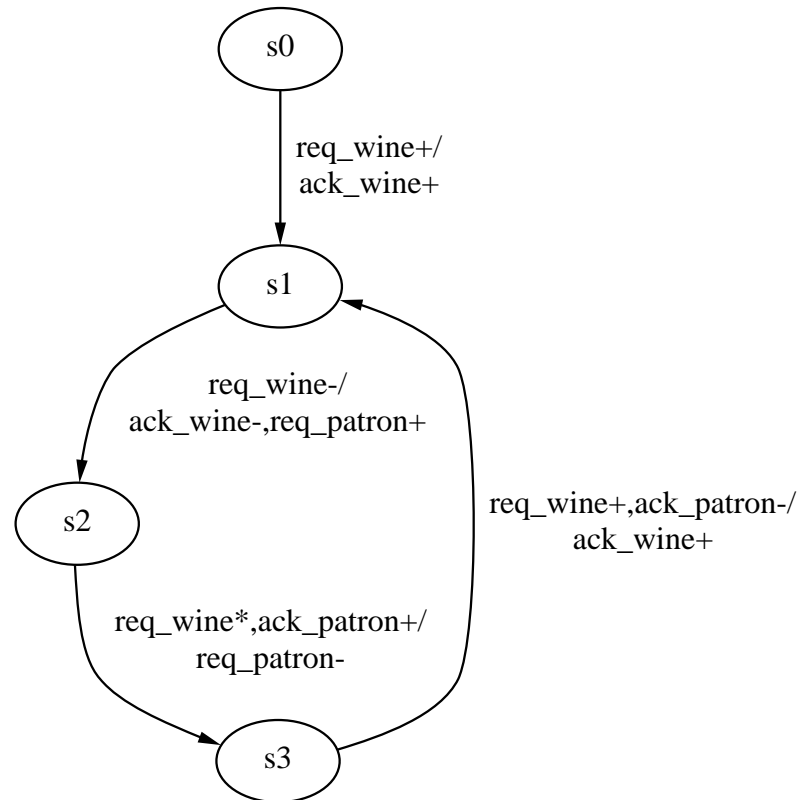
## Extended Burst-Mode

- 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

```
Shop_PA_2: process
begin
    guard(req_wine, '1');          -- winery calls
    assign(ack_wine, '1', 1, 3);   -- receives wine
    guard(req_wine, '0');          -- req_wine reset
    assign(ack_wine = '0', 1, 3, req_patron, '1', 1, 3);
    guard(ack_patron, '1');        -- wine purchased
    assign(req_patron, '0', 1, 3); -- reset req_patron
    guard(ack_patron, '0');        -- ack_patron reset
end process;
```

# Directed Don't Cares



## Directed Don't Cares

*req\_wine / ack\_patron*

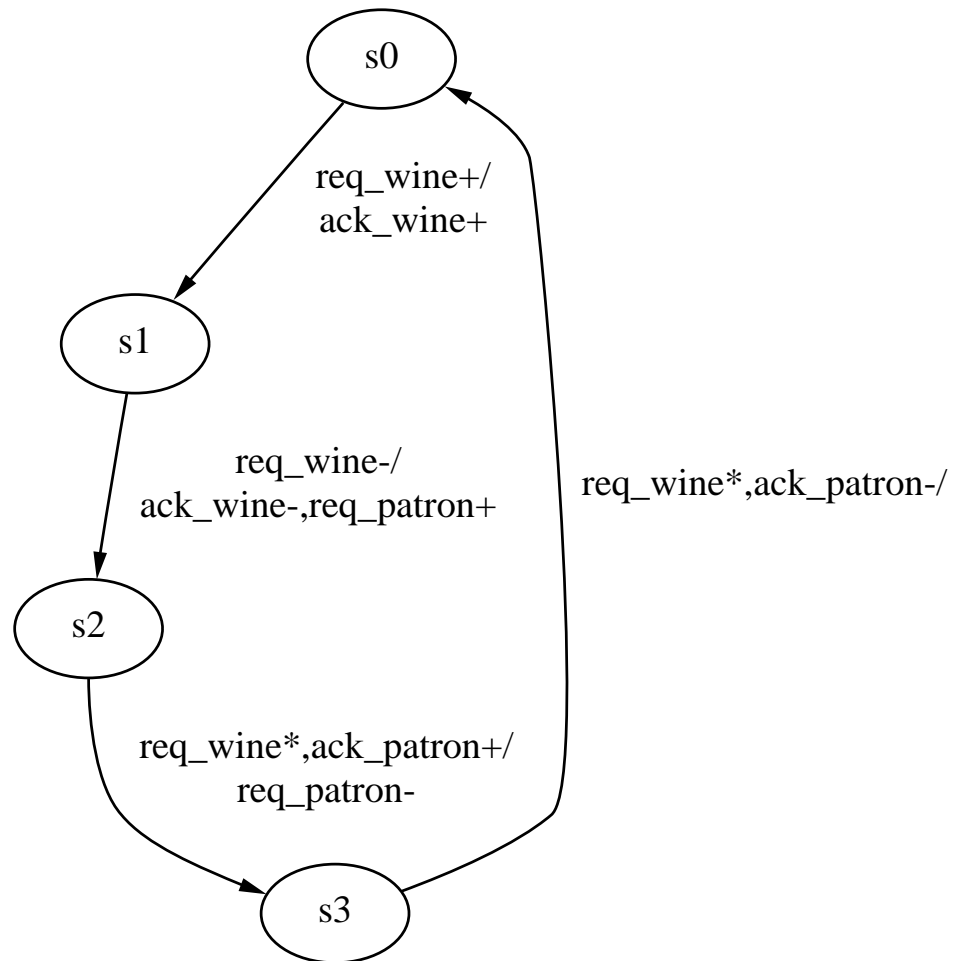
	00	01	11	10
s0	⓪, 00	—	—	s1, 10
s1	s2, 01	—	—	Ⓛ1, 10
s2	Ⓜ2, 01	s3, 00	s3, 00	Ⓜ2, 01
s3	Ⓜ3, 10	Ⓜ3, 10	Ⓜ3, 10	s1, 10

*ack\_wine / req\_patron*

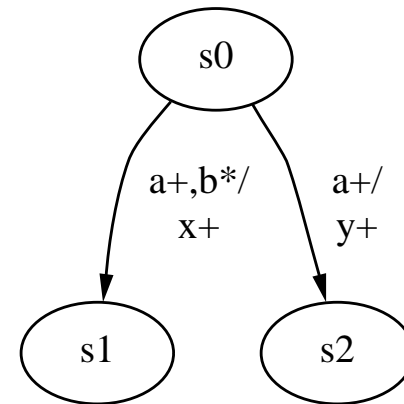
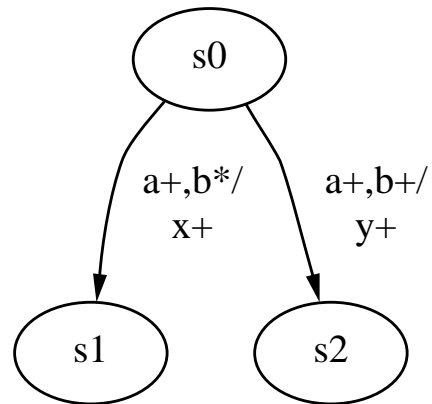
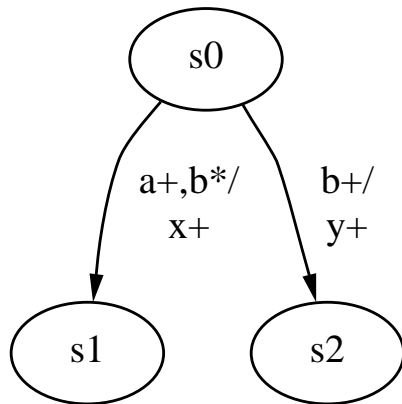
## Directed Don't Cares

- 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



## Modified Maximal Set Property



## Conditional Input Bursts

```
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');
```

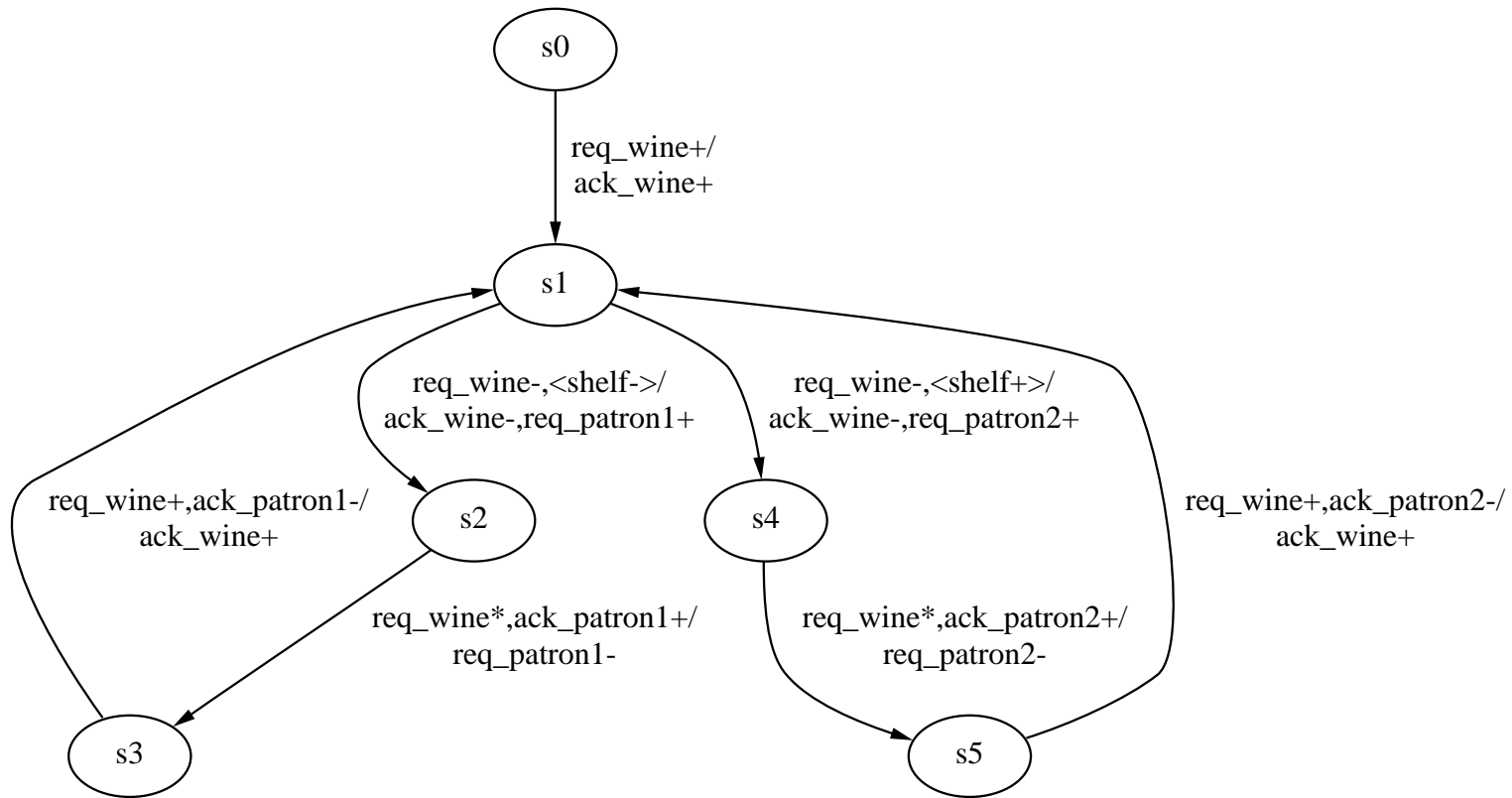
## Conditional Input Bursts

```
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;
```

## Conditional Input Bursts

- A conditional input burst includes a regular input burst and a *conditional clause*.
- A clause of the form  $\langle s- \rangle$  indicates that the transition is only taken if  $s$  is low.
- A clause of the form  $\langle s+ \rangle$  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



## Conditional Input Bursts

	<i>req_wine / ack_patron1 / ack_patron2 / shelf</i>							
	0000	0001	0011	0010	0110	0111	0101	0100
s0	(s0) 000	(s0) 000	—	—	—	—	—	—
s1	s2, 010	s4, 001	—	—	—	—	—	—
s2	(s2) 010	(s2) 010	—	—	—	—	s3, 000	s3, 000
s3	(s3) 010	(s3) 010	—	—	—	—	(s3) 000	(s3) 000
s4	(s4) 001	(s4) 001	s5, 000	s5, 000	—	—	—	—
s5	(s5) 000	(s5) 000	(s5) 000	(s5) 000	—	—	—	—

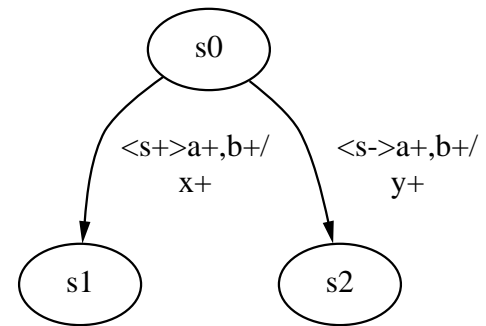
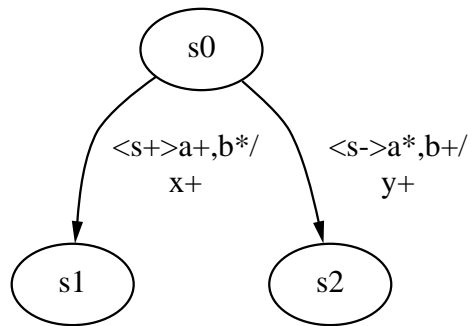
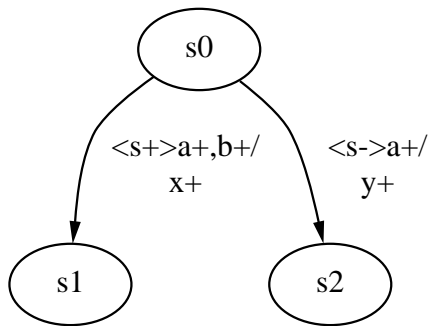
*ack\_wine / req\_patron1 / req\_patron2*

## Conditional Input Bursts

	<i>req_wine / ack_patron1 / ack_patron2 / shelf</i>							
	1100	1101	1111	1110	1010	1011	1001	1000
s0	—	—	—	—	—	—	s1, 100	s1, 100
s1	—	—	—	—	—	—	(s1) 100	(s1) 100
s2	s3, 000	s3,000	—	—	—	—	(s2) 010	(s2) 010
s3	(s3) 000	(s3) 000	—	—	—	—	s1, 100	s1, 100
s4	—	—	—	—	s5, 000	s5,000	(s4) 001	(s4) 001
s5	—	—	—	—	(s5) 000	(s5) 000	s1, 100	s1, 100

*ack\_wine / req\_patron1 / req\_patron2*

## Modified Maximal Set Property



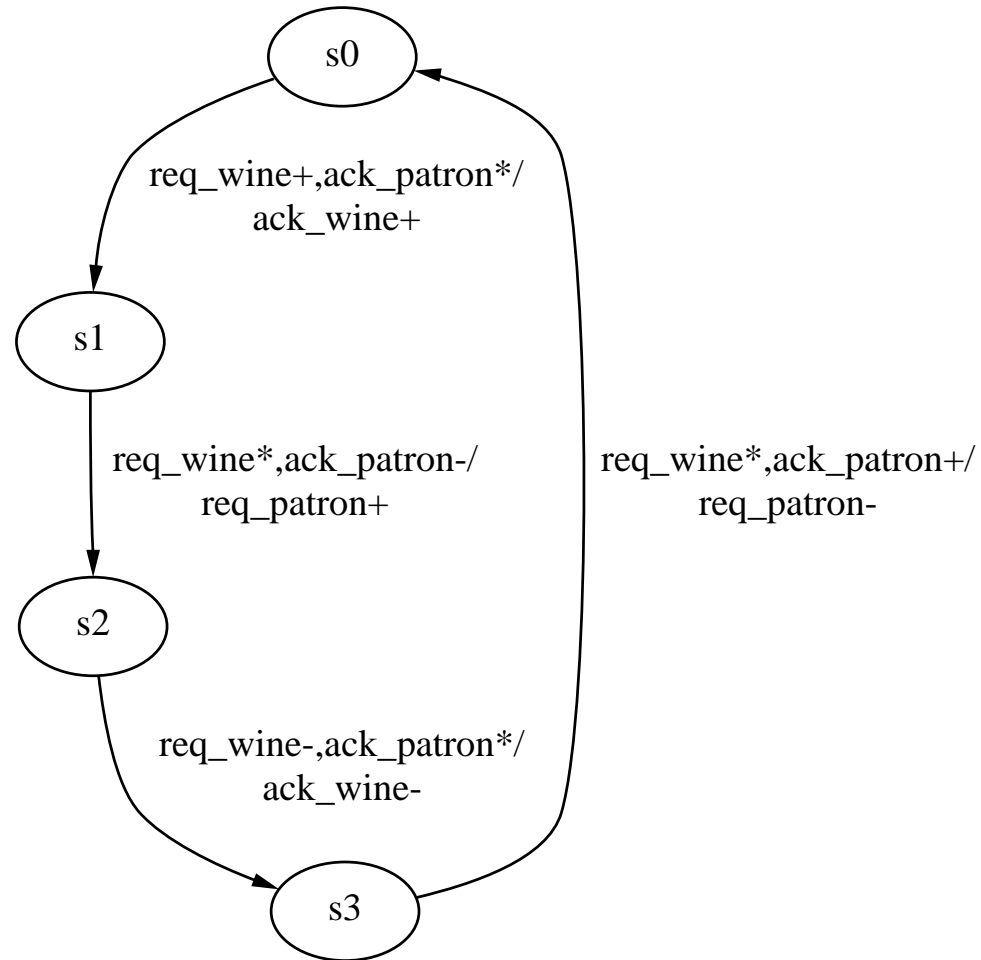
## Burst-Mode State Machines

- $V$  is a finite set of vertices (or states).
- $E \subseteq V \times V$  is the set of edges (or transitions).
- $I = \{x_1, \dots, x_m\}$  is the set of inputs.
- $O = \{z_1, \dots, z_n\}$  is the set of outputs.
- $C = \{c_1, \dots, c_l\}$  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.

## No XBM Machine

```
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;
```

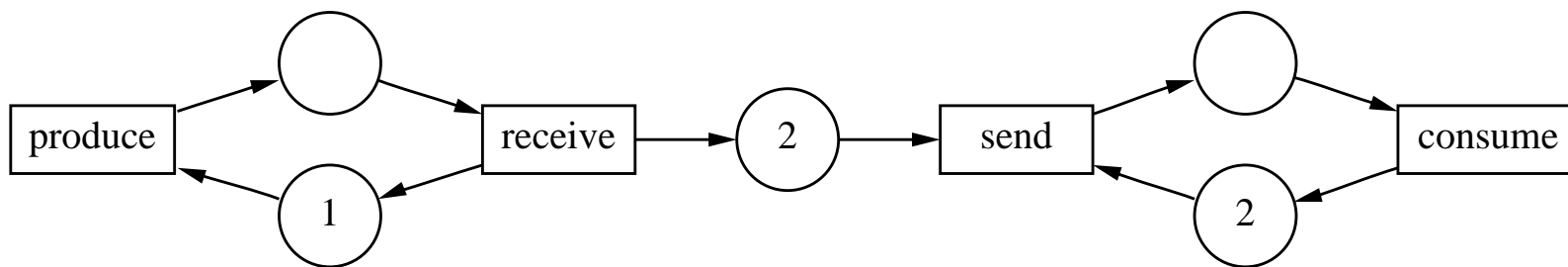
# Illegal XBM Machine



## Petri-Nets

- 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



## Preset and Postset

- 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.,  $p\bullet = \{t \in T \mid (p, t) \in F\}$ ).

## Markings

- 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:

$$M \geq M' \quad \text{iff} \quad \forall p \in P . M(p) \geq M'(p)$$

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

$$C_A(p) = \text{if } p \in A \text{ then } 1 \text{ else } 0.$$

## Transition Firings

- 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  $M[t\rangle M'$ ):

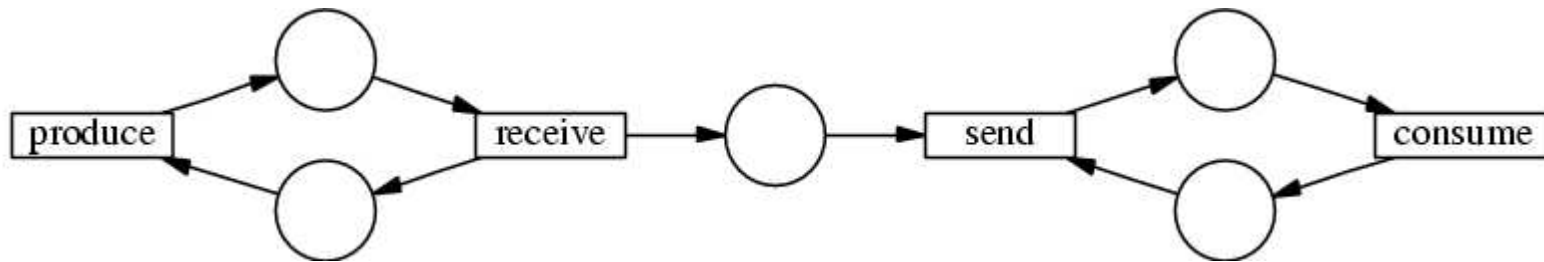
$$M' = M - C_{\bullet t} + C_{t\bullet}$$

- 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.

## Reachable Markings

- Firing of a transition transforms the marking of the Petri net into a new marking.
- A sequence of transition firings ( $\sigma = t_1, t_2, \dots, t_n$ ) produces a sequence of markings ( $M_0, M_1, \dots, M_n$ ).
- If such a *firing sequence* exists, we say that the marking  $M_n$  is *reachable* from  $M_0$  by  $\sigma$  (denoted  $M_0[\sigma\rangle M_n$ ).
- We denote the set of all markings reachable from a given marking by  $[M\rangle$ .

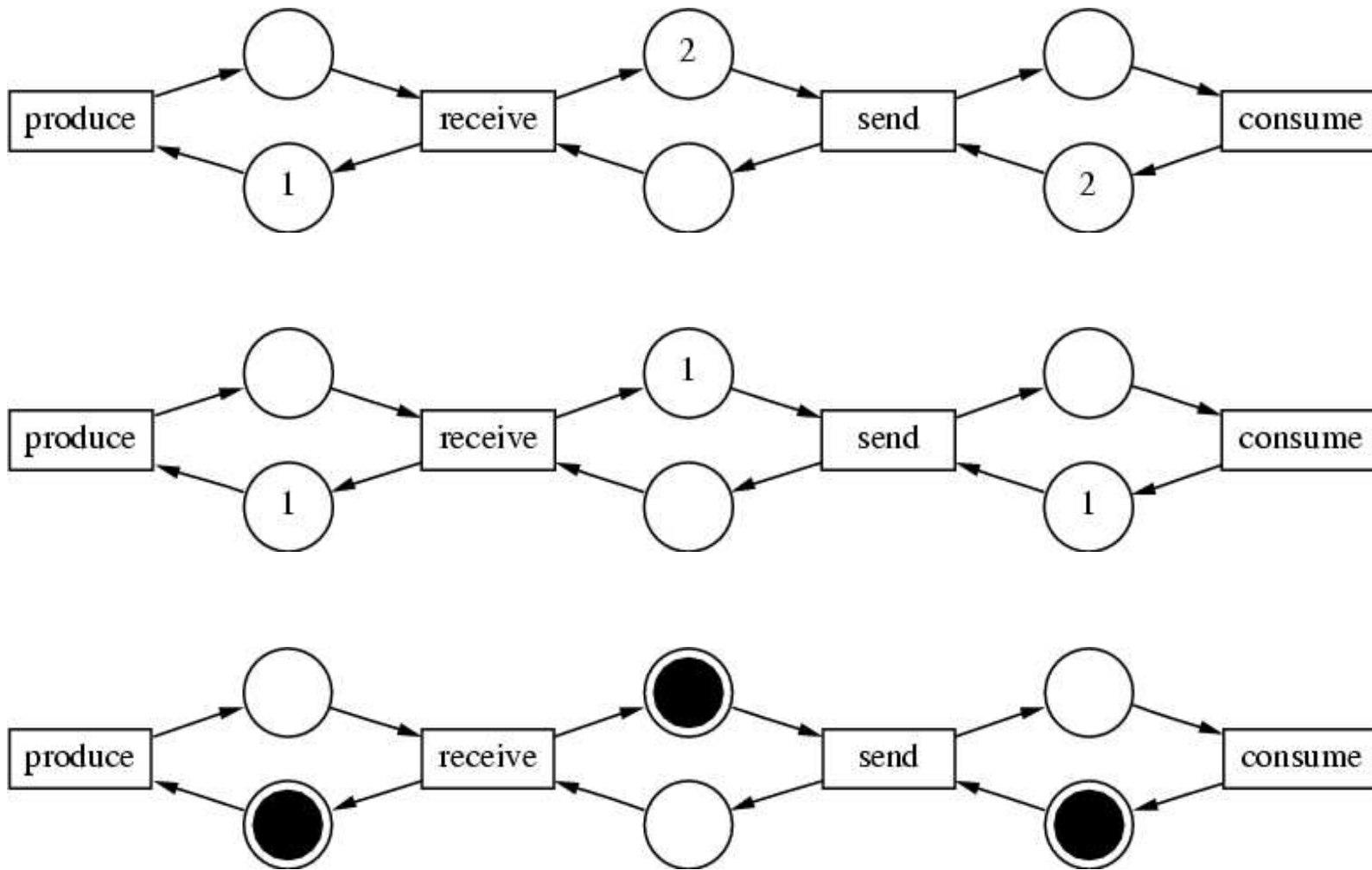
## Example Firing Sequence



## **$k$ -Bounded Petri-Nets**

- 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



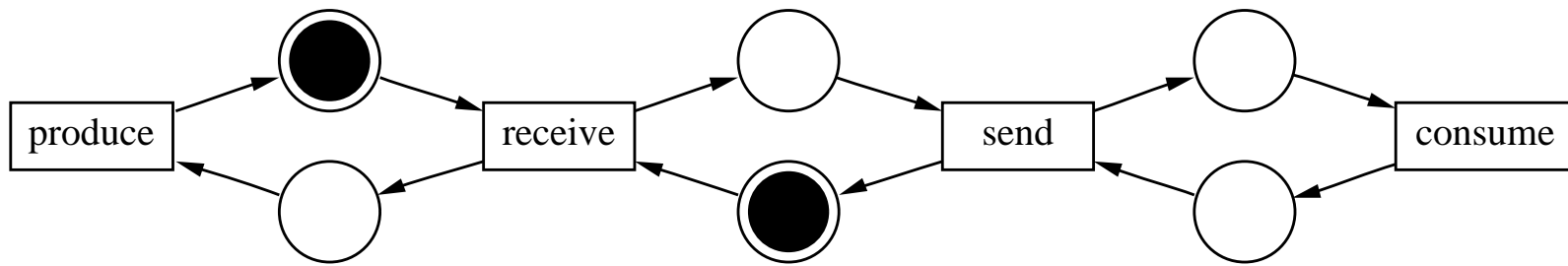
## Liveness

- A Petri net is *live* if from every reachable marking, there exists a sequence of transitions such that any transition can fire.

$$\forall M \in [M_0\rangle, \forall t \in T, \exists M' \in [M\rangle. M' \geq C_{\bullet t}$$

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

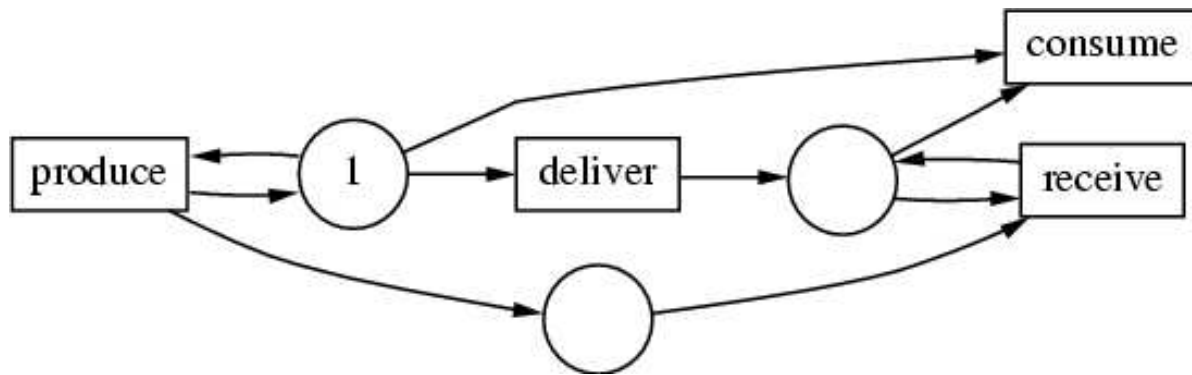
# Liveness



## Liveness Categories

- 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



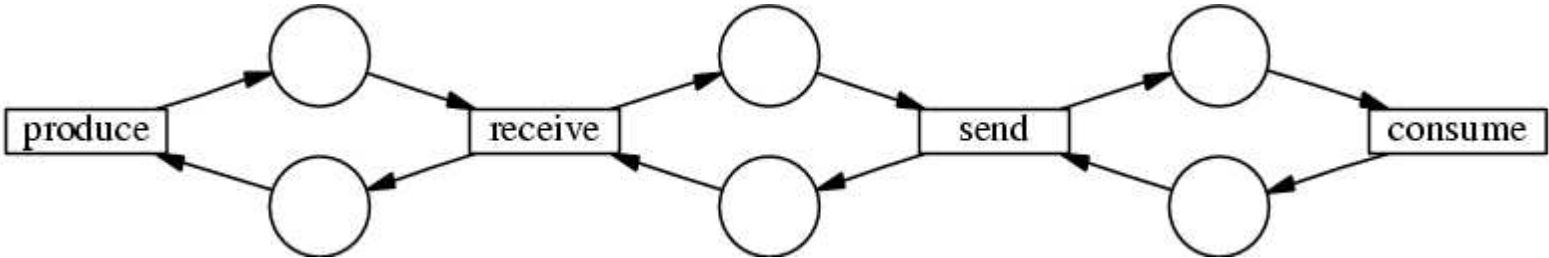
## Reachability Graph

- When a Petri net is bounded, the number of reachable markings is finite, and a *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.

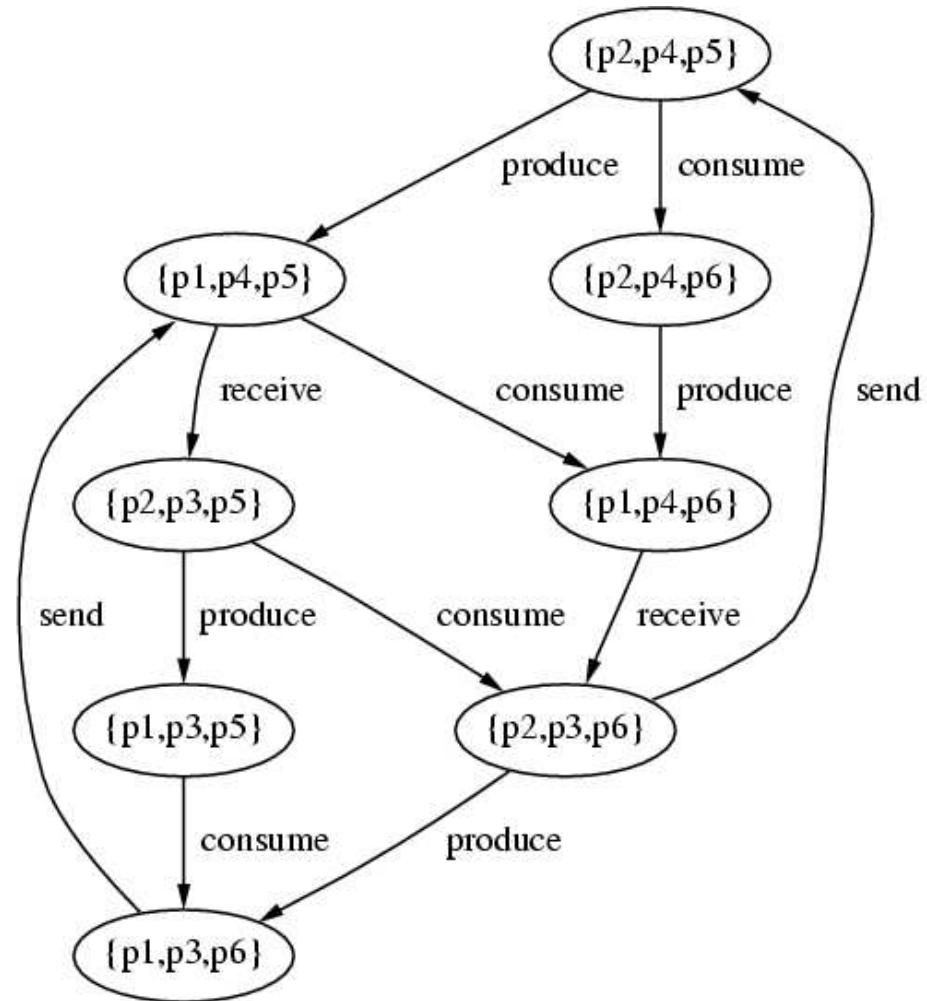
## Algorithm to Find Reachability Graph

```
find_RG(Petri net  $\langle P, T, F, M_0 \rangle$ ) {  
   $M = M_0$ ;  $T_e = \{t \in T \mid M \geq C_{\bullet t}\}$ ;  $\Phi = \{M\}$ ;  $\Gamma = \emptyset$ ;  
  done = false;  
  while ( $\neg$  done) {  
     $t = \text{select}(T_e)$ ;  
    if ( $T_e - \{t\} \neq \emptyset$ ) then push( $M, T_e - \{t\}$ );  
     $M' = M - C_{\bullet t} + C_{t\bullet}$ ;  
    if ( $M' \notin \Phi$ ) then {  
       $\Phi = \Phi \cup \{M'\}$ ;  $\Gamma = \Gamma \cup \{(M, M')\}$ ;  
       $M = M'$ ;  $T_e = \{t \in T \mid M \geq C_{\bullet t}\}$ ;  
    } else {  
       $\Gamma = \Gamma \cup \{(M, M')\}$ ;  
      if (stack is not empty) then ( $M, T_e$ ) = pop();  
      else done = true; } }  
  return( $\Phi, \Gamma$ ); }
```

# Safe Example



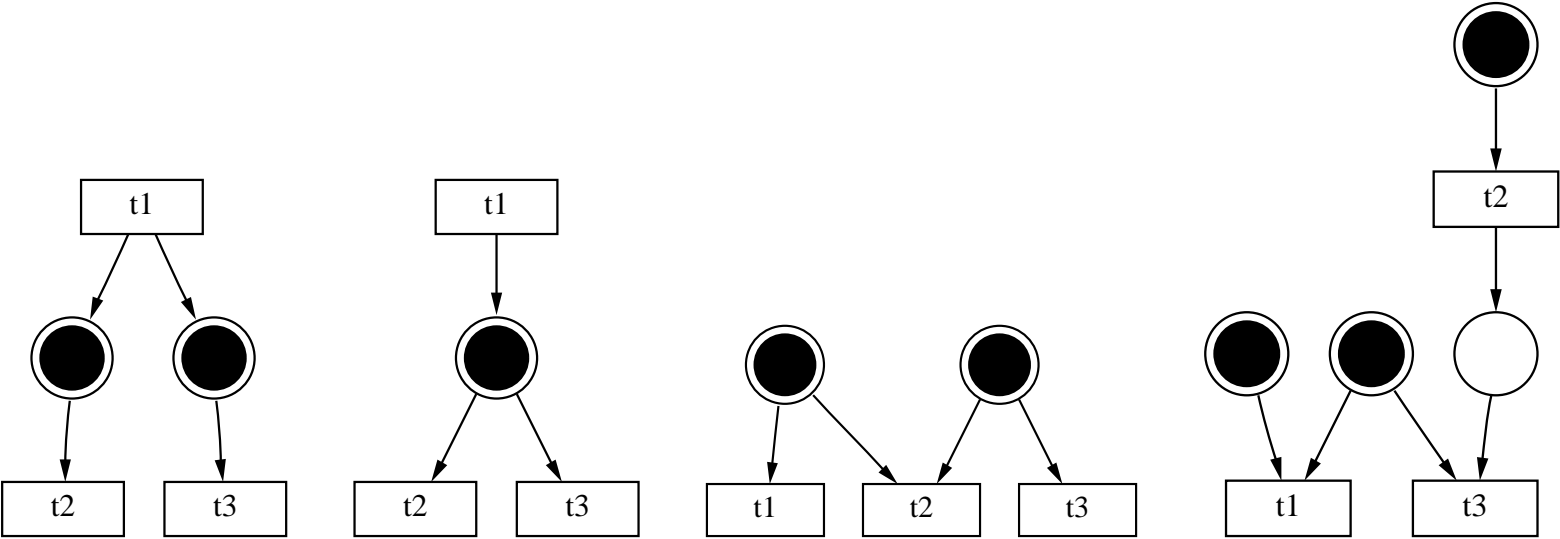
## Example Reachability Graph



## Concurrency, Conflict, and Confusion

- Two transitions  $t_1$  and  $t_2$  are *concurrent* when there exists markings where both enabled and can fire in either order.
- Two transitions,  $t_1$  and  $t_2$ , are in *conflict* when the firing of one disables the other.
- When concurrency and conflict are mixed, we get *confusion*.

# Example of Concurrency, Conflict, and Confusion



## State Machines and Marked Graphs

- 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.

## Free Choice Nets

- 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.

$$\forall t, t' \in T, t \neq t' : \bullet t \cap \bullet t' \neq \emptyset \Rightarrow |\bullet t| = |\bullet t'| = 1$$

$$\forall p, p' \in P, p \neq p' : p \bullet \cap p' \bullet \neq \emptyset \Rightarrow |p \bullet| = |p' \bullet| = 1$$

$$\forall p \in P, \forall t \in T : (p, t) \in F \Rightarrow p \bullet = \{t\} \vee \bullet t = \{p\}$$

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



## Extended Free Choice Nets

- 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.

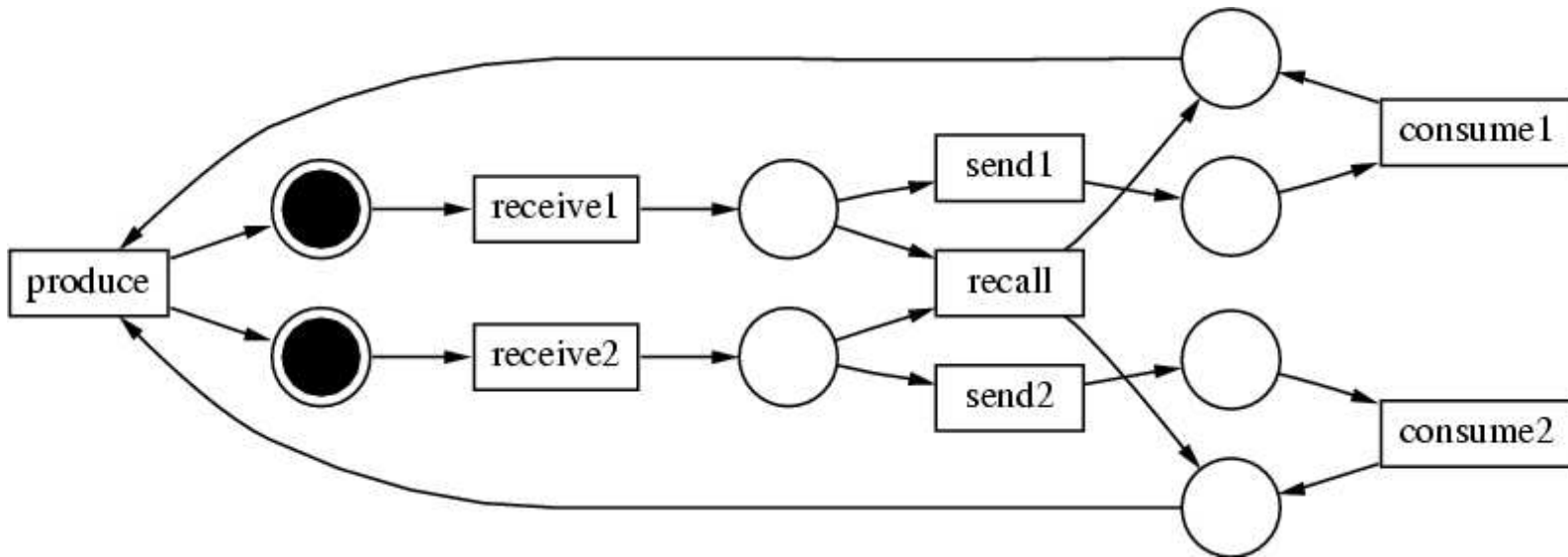
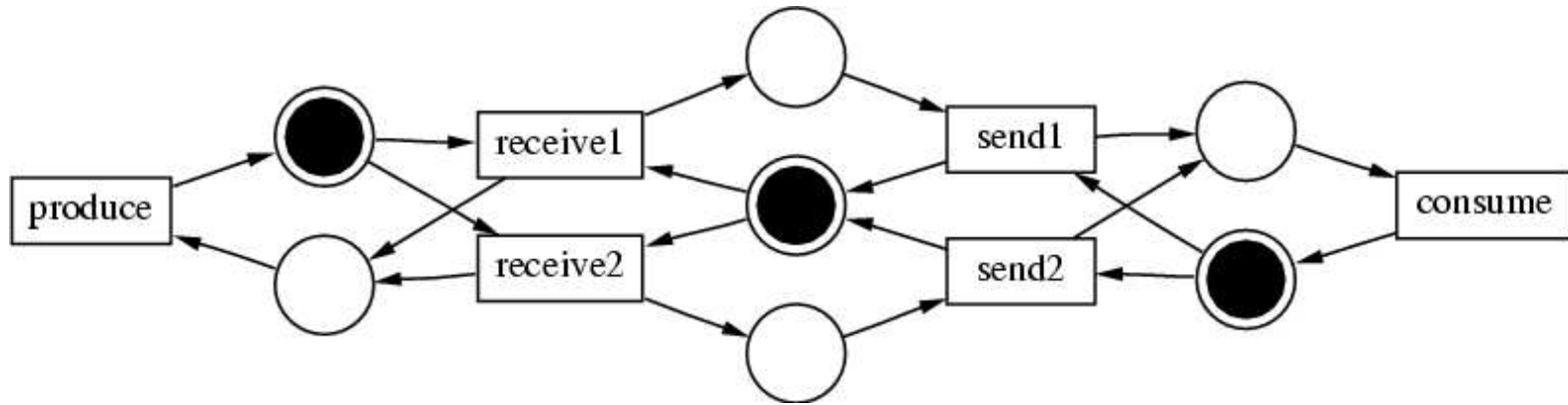
## Asymmetric Choice Nets

- 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.

$$\forall p, p' \in P . p \bullet \cap p' \bullet \neq \emptyset \Rightarrow p \bullet \subseteq p' \bullet \vee p' \bullet \subseteq p \bullet$$

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

## More Example Nets



## Checking Safety and Liveness

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

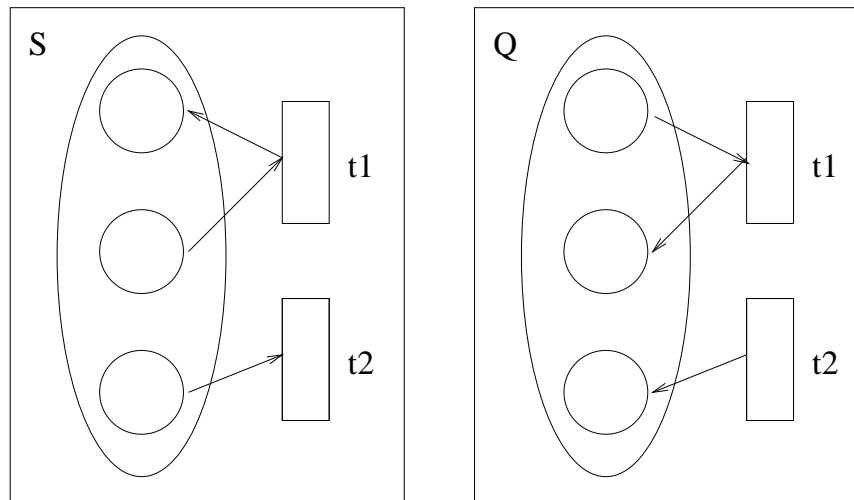
**Theorem 4.1** A state machine is live and safe iff it is strongly connected and  $M_0$  has exactly one token.

**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.

## Siphons and Traps

- 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



## Checking Liveness

**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  $M_0$ .

## Marked Graph Components

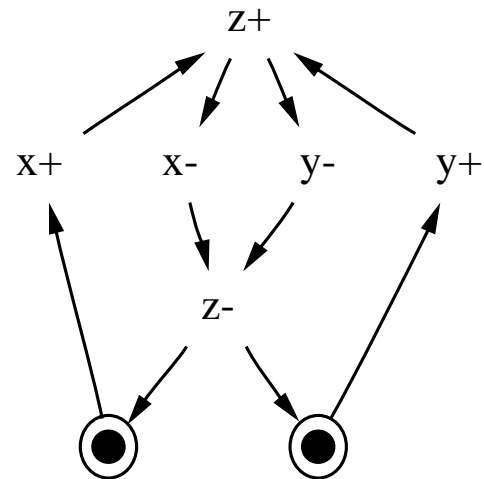
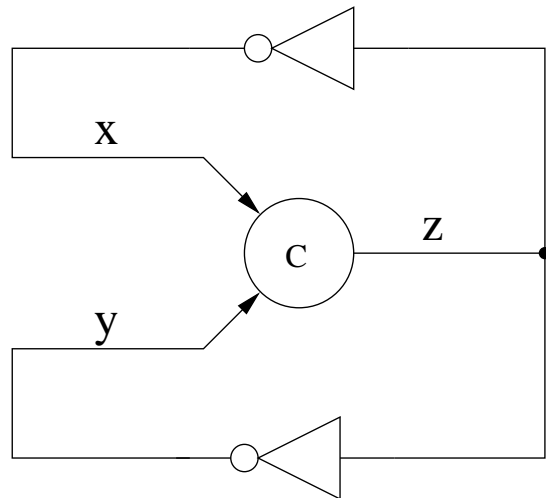
- A *marked graph component* of a net,  $N$ , is a subnet in which each place has at most one transition in its preset and one transition in its postset and is generated by these transitions.
- The net generated by a set of transitions includes these transitions, all places in their preset and postset, 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.

## Signal Transition Graphs (STG)

- 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.

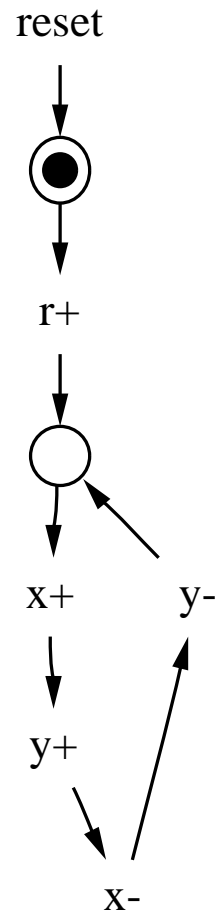
## Example Signal Transition Graph (STG)



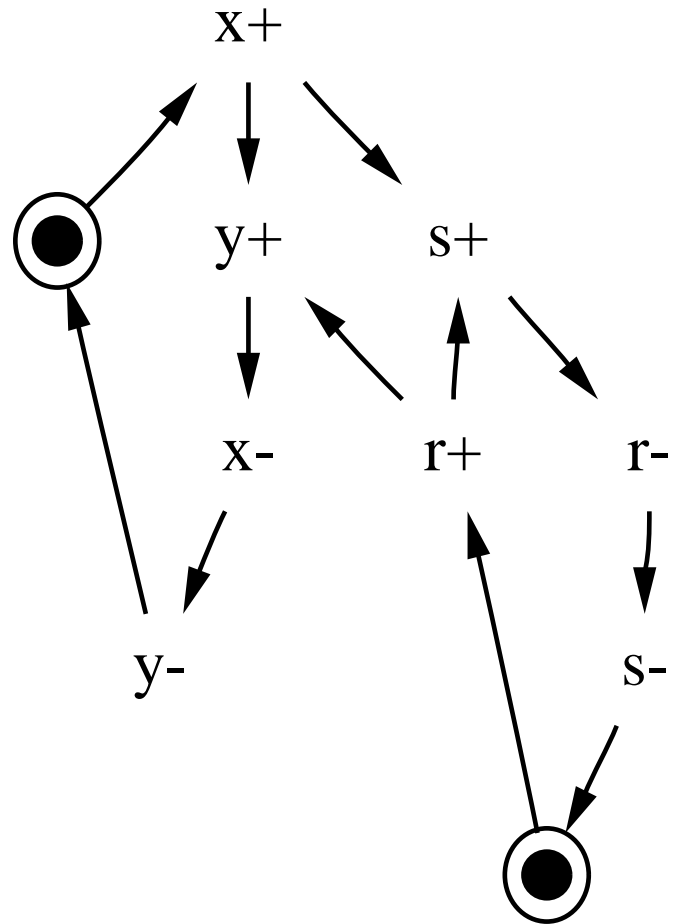
## STG Restrictions

- 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.

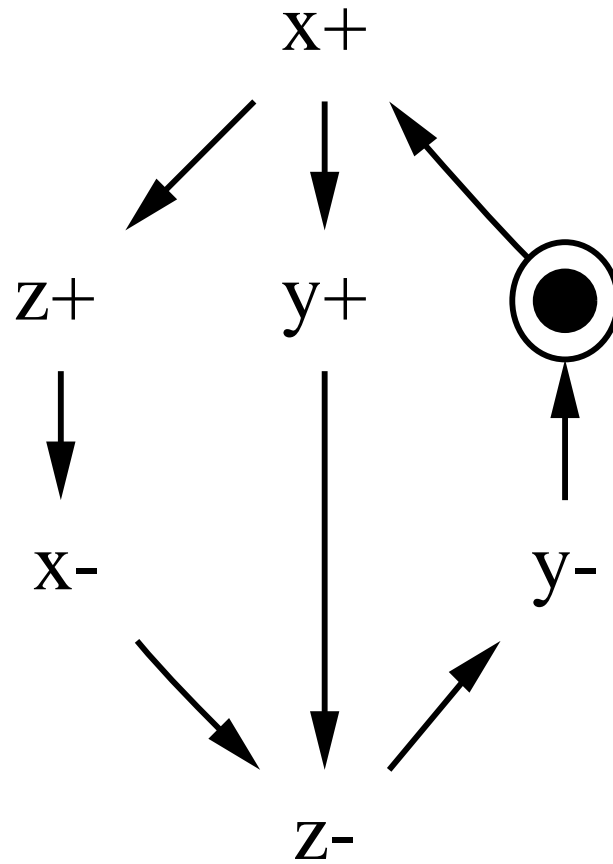
# Liveness



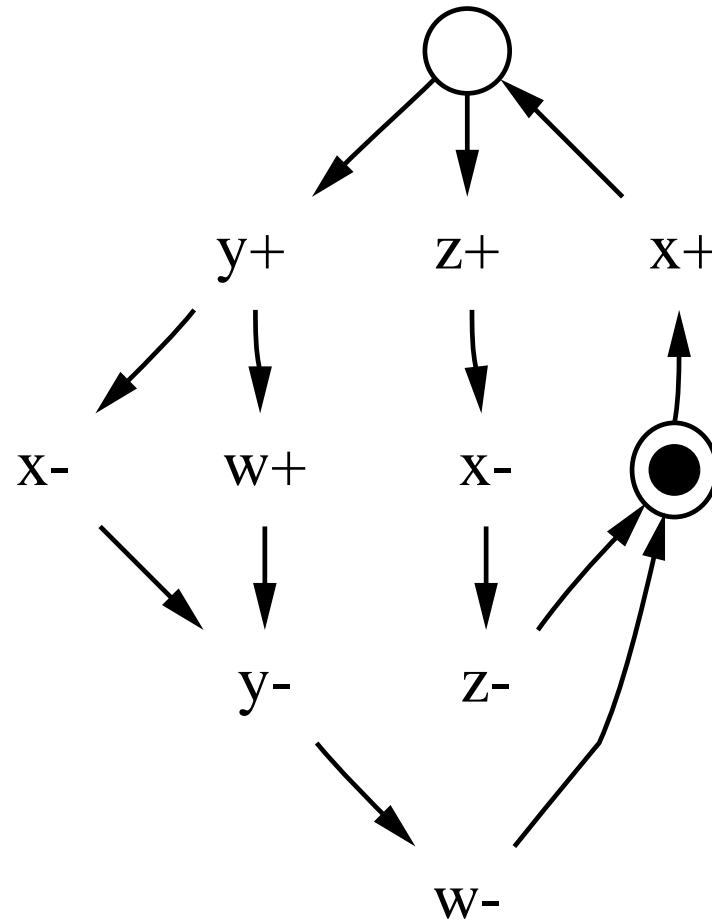
# Safety



# Persistency



# Single-Cycle Transitions



## State Graphs (SG)

- 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  $\langle s(0), s(1), \dots, s(n) \rangle$ , where  $s(i)$  is either 0 or 1, indicating value returned by  $\lambda_S$ .
- We use  $s(i)$  interchangeably with  $\lambda_S(s)(i)$ .

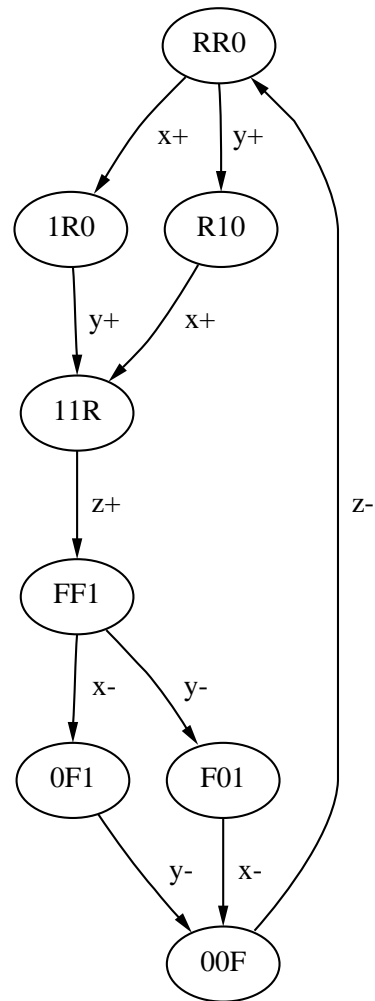
## Implied State

- If in  $s_i$ , there exists a transition on signal  $u_i$  to  $s_j$  [i.e.,  $\exists(s_i, t, s_j) \in \delta \cdot \lambda_T(t) = u_i + \vee \lambda_T(t) = u_i -$ ], 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  $\overline{s(i)}$ .
- 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  $\langle s'(0), s'(1), \dots, s'(n) \rangle$  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”.

## Algorithm to Find SG

```
find_SG( $\langle P, T, F, M_0, N, s_0, \lambda_T \rangle$ ) {  
   $M = M_0$ ;  $s = s_0$ ;  $S = \{M\}$ ;  $\lambda_S(M) = s$ ;  
   $T_e = \{t \in T \mid M \subseteq \bullet t\}$ ; done = false;  
  while ( $\neg$  done) {  
     $t = \text{select}(T_e)$ ;  
    if ( $T_e - \{t\} \neq \emptyset$ ) then push( $M, s, T_e - \{t\}$ );  
    if ( $(M - \bullet t) \cap t\bullet \neq \emptyset$ ) then return(“Not safe.”);  
     $M' = (M - \bullet t) \cup t\bullet$ ;  $s' = s$ ;  
    if ( $\lambda_T(t) = u+$ ) then  $s'(u) = 1$ ;  
    else if ( $\lambda_T(t) = u-$ ) then  $s'(u) = 0$ ;  
    if ( $M' \notin S$ ) then {  
       $S = S \cup \{M'\}$ ;  $\lambda_S(M') = s'$   $\delta = \delta \cup \{(M, t, M')\}$ ;  
       $M = M'$ ;  $s = s'$ ;  $T_e = \{t \in T \mid M \subseteq \bullet t\}$ ;  
    } else {  
      if ( $\lambda_S(M') \neq s'$ ) then return(“Inconsistent.”);  
      if (stack is not empty) then  $(M, s, T_e) = \text{pop}()$ ;  
      else done = true; } }  
  return( $\langle S, \delta, \lambda_S \rangle$ ); }
```

# SG for C-Element



## Consistent State Assignment

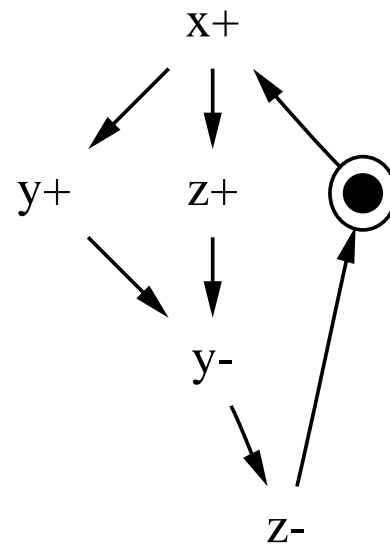
- A well-formed SG, must have a *consistent state assignment*.
- A SG has a consistent state assignment if for each state transition  $(s_i, t, s_j) \in \delta$  exactly one signal changes value, and its value is consistent with the transition.

$$\begin{aligned} \forall (s_i, t, s_j) \in \delta. \forall u \in N \quad & \cdot \quad (\lambda_T(t) \neq u * \wedge s_i(u) = s_j(u)) \\ & \vee \quad (\lambda_T(t) = u + \wedge s_i(u) = 0 \wedge s_j(u) = 1) \\ & \vee \quad (\lambda_T(t) = u - \wedge s_i(u) = 1 \wedge s_j(u) = 0) \end{aligned}$$

where “\*” represents either “+” or “-”.

- A STG produces a SG with a consistent state assignment if in any firing sequence the transitions of a signal strictly alternate between +’s and -’s.

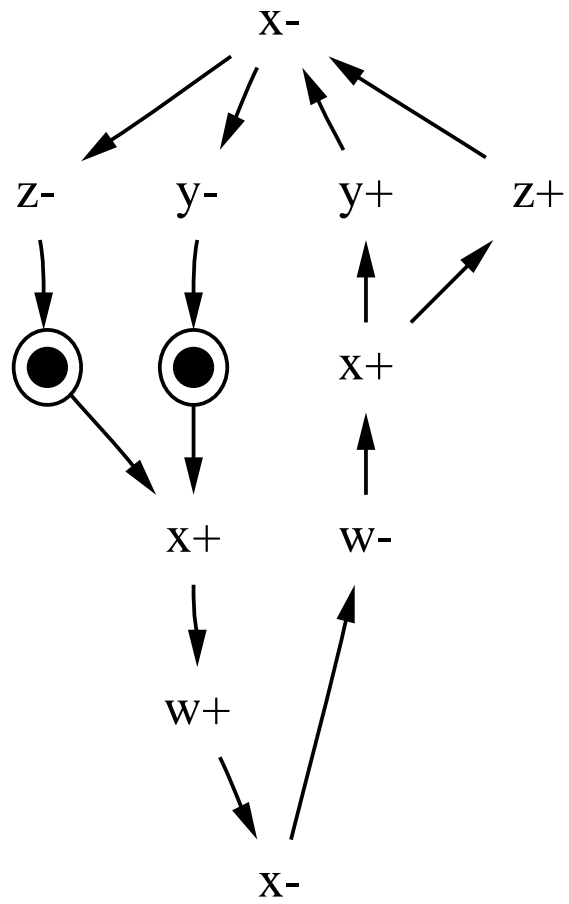
## Consistent State Assignment



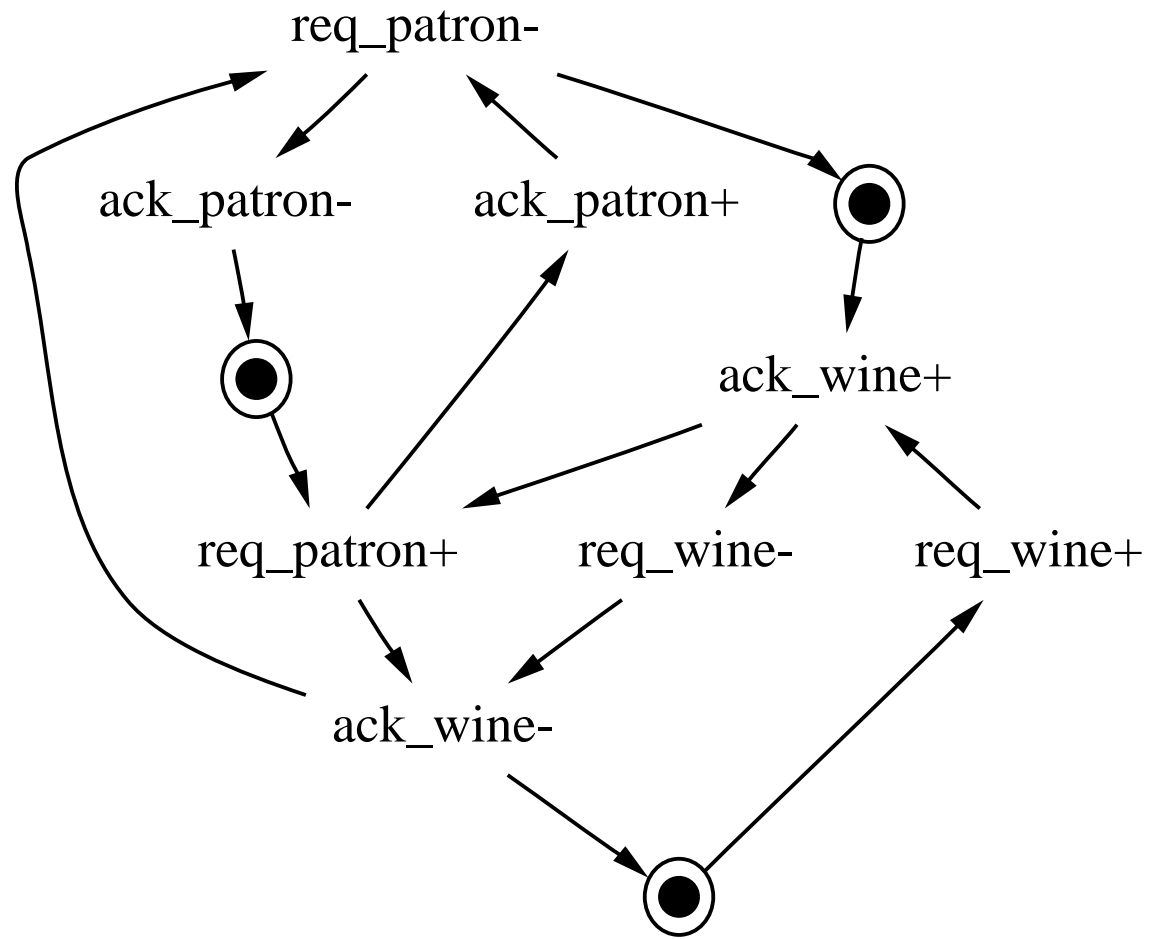
## Unique State Code

- A SG has a *unique state assignment* (USC) if no two different states (i.e., markings) have identical values for all signals [i.e.,  $\forall s_i, s_j \in S, s_i \neq s_j . \lambda(s_i) \neq \lambda(s_j)$ ].
- Some synthesis methods are restricted to STGs that produce SGs with USC.

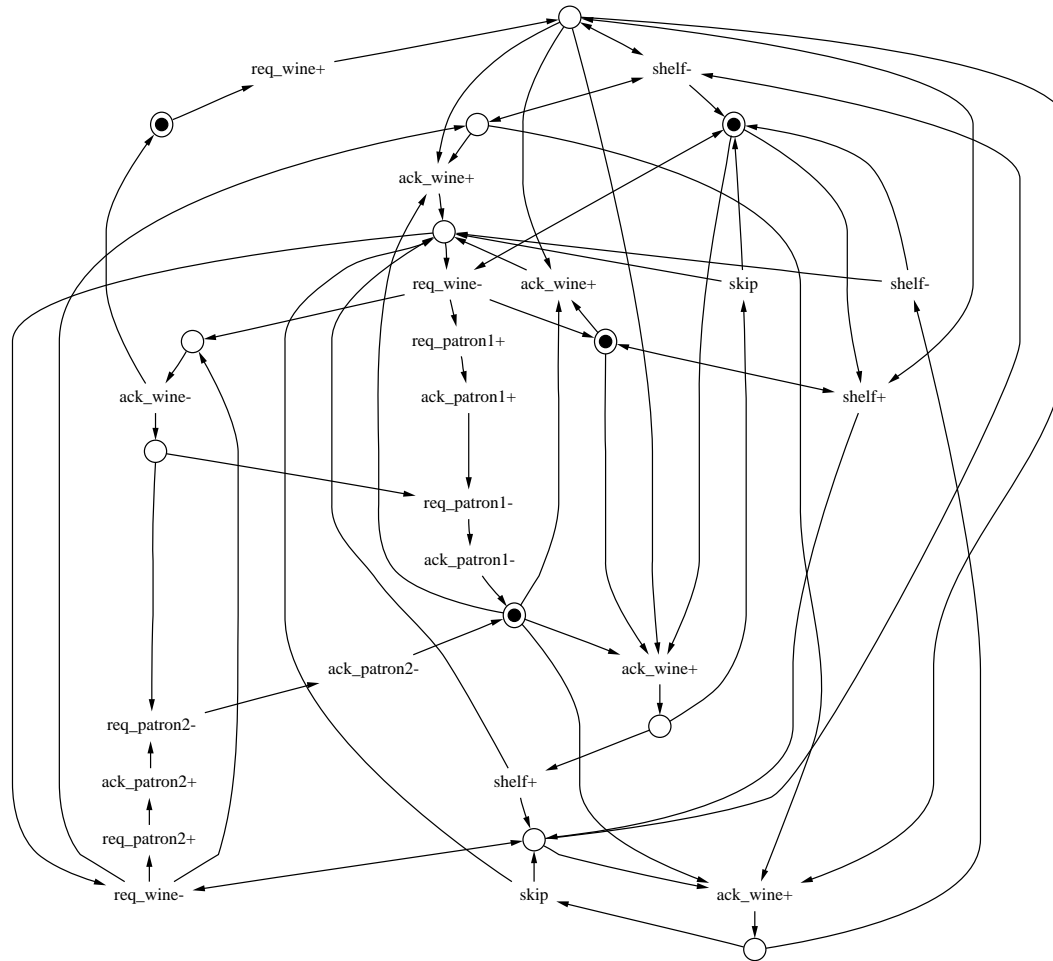
# Unique State Code



# STG for Reshuffled Passive/Lazy-Active Wine Shop



# STG for Wine Shop with Two Patrons



## Timed Event/Level (TEL) Structures

- AFSMs cannot model arbitrary concurrency.
- Petri-nets have difficulty to express signal levels.
- *Timed event/level (TEL) structures* are a hybrid graphical representation method which are both capable of modelling arbitrary concurrency and signal levels.

## Timed Event/Level (TEL) Structures

- $N$  is the set of signals.
- $s_0 = \{0, 1\}^N$  is the initial state.
- $A \subseteq N \times \{+, -\} \cup \$$  is the set of atomic *actions*.
- $E \subseteq A \times (\mathcal{N} = \{0, 1, 2, \dots\})$  is the set of *events*.
- $R \subseteq E \times E \times \mathcal{N} \times (\mathcal{N} \cup \{\infty\}) \times (b: \{0, 1\}^N \rightarrow \{0, 1\})$  is the set of *rules*.
- $R_O \subseteq R$  is the set of *initially marked rules*.
- $\# \subseteq E \times E$  is the *conflict relation*.

## Rules

- $e = \textit{enabling event}$ .
- $f = \textit{enabled event}$ .
- $[l, u] = \textit{bounded timing constraint}$ .
- $b = \textit{a boolean function over the signals in } N$ .

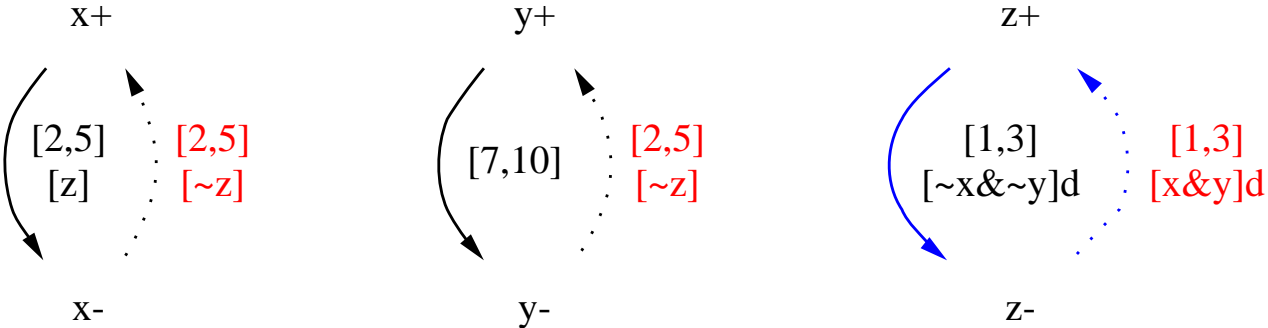
## Rule Timing

- A rule is *enabled* if its enabling event has occurred and its boolean function is true in the current state.
- A rule is *satisfied* if it has been enabled at least  $l$  time units.
- A rule becomes *expired* when it has been enabled  $u$  time units.
- Excluding conflicts, an event cannot occur until every rule enabling it is satisfied, and it must occur before every rule enabling it has expired.

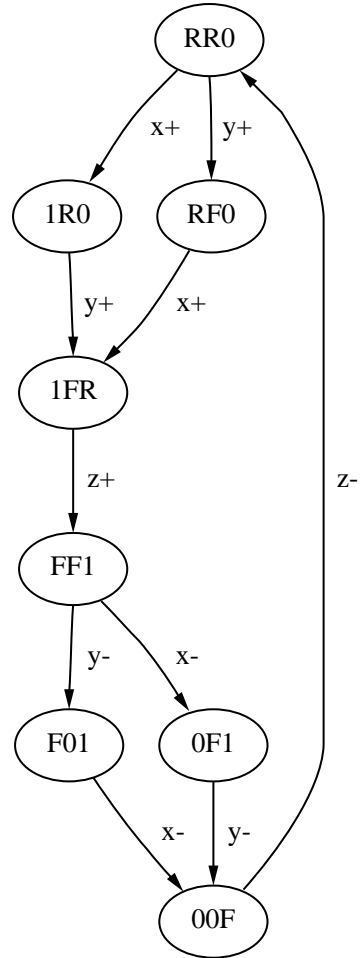
## Disabling versus Nondisabling Rules

- Special care must be taken when a rule becomes disabled.
- Each rule is defined to be *disabling* or *nondisabling*.
- If a rule is disabling and its boolean condition becomes false, it ceases to be enabled until condition is true again.
- If a rule is nondisabling, the disabling of the boolean condition is ignored.
- For verification, if a rule is disabled, this may indicate that the enabled event has a hazard which is considered a failure.

# TEL Structure for a C-Element



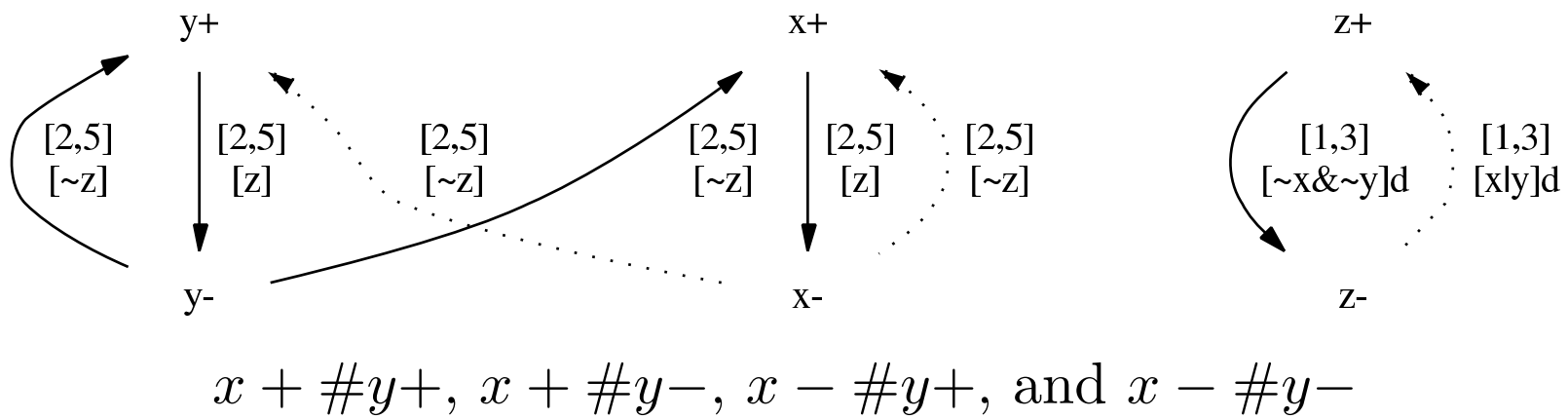
## SG for a Timed C-Element



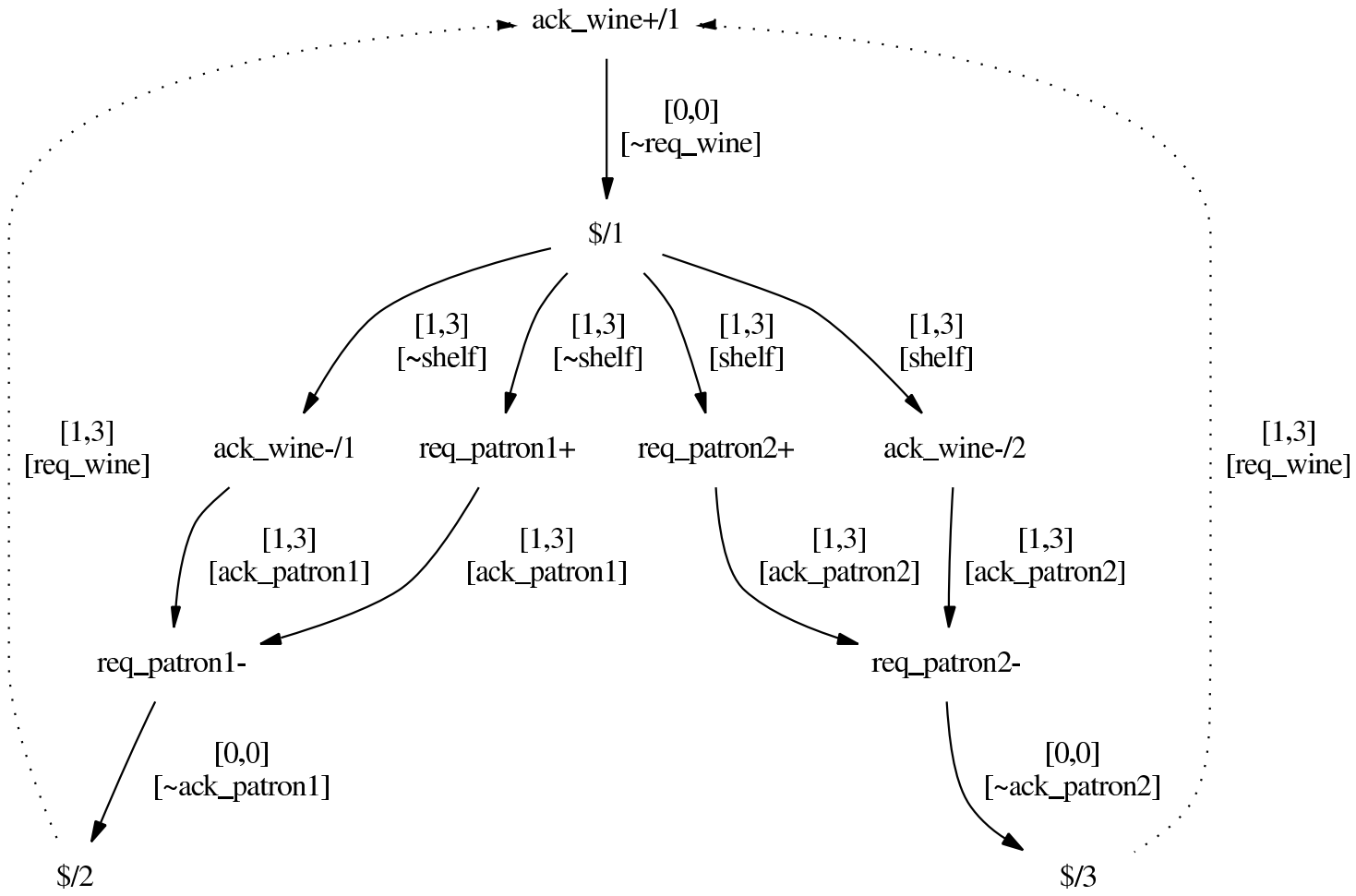
## Conflict Relation

- The conflict relation,  $\#$ , is used to model disjunctive behavior and choice.
- When two events  $e$  and  $e'$  are in conflict (denoted  $e\#e'$ ), this specifies that either  $e$  or  $e'$  can occur but not both.
- If two rules have the same enabled event and conflicting enabling events, then only one of the two mutually exclusive enabling events needs to occur to cause the enabled event.
- When two rules have the same enabling event and conflicting enabled events, only one of the enabled events can occur.

## TEL Structure with Conflict



# TEL Structure for Wine Shop with Two Patrons



## Summary

- Finite state machines (AFSMs, BM, and XBM).
- Petri-nets and STGs.
- TEL structures.