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

Slide 4
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.

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.
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.

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 $(V, R, L)$ in which $L$ is a labeling function associated either to the set of vertices or edges.
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;
- $\delta : S \times I \to S$ is the next-state function;
- $\lambda : S \times I \to O$ is the output function for a Mealy machine (or $\lambda : S \to O$ 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., \((u, v) \in E \iff \exists i \in I \text{ s.t. } ((u, i), v) \in \delta \)).
- 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 \(((u, i), o) \in \lambda\).

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(ack_patron,'0',1,3); -- reset req_patron
  guard(ack_wine,'0');   -- ack_wine reset
  assign(ack_wine,'0',1,3); -- reset ack_wine
end process;
```

Passive/Active Shop FSM

```
\begin{array}{c|ccccc}
\text{req_wine} / \text{ack_patron} & 00 & 01 & 11 & 10 \\
\hline
s0 & \emptyset & 00 & - & - & s1, 10 \\
s1 & s2, 11 & - & - & \emptyset & 10 \\
s2 & \emptyset & 11 & s3, 10 & - & - \\
s3 & s0, 00 & \emptyset & 10 & - & - \\
\end{array}
```

Passive/Active Shop FSM

```
\begin{array}{c|ccccc}
\text{req_wine} / \text{ack_patron} & 00 & 01 & 11 & 10 \\
\hline
s0 & \emptyset & 00 & - & - & s1, 10 \\
s1 & s2, 11 & - & - & \emptyset & 10 \\
s2 & \emptyset & 11 & s3, 10 & - & - \\
s3 & s0, 00 & \emptyset & 10 & - & - \\
\end{array}
```

Burst-Mode State Machine

```
\begin{array}{c|ccccc}
\text{req_wine} / \text{ack_patron} & 00 & 01 & 11 & 10 \\
\hline
s0 & \emptyset & 00 & - & - & s1, 10 \\
s1 & s2, 11 & - & - & \emptyset & 10 \\
s2 & \emptyset & 11 & s3, 10 & - & - \\
s3 & s0, 00 & \emptyset & 10 & - & - \\
\end{array}
```
### 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;
- $v_0 \in V$ 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.

### 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.
- $\forall (u, v), (u, w) \in E : trans_s(u, v) \subseteq trans_s(u, w) \Rightarrow v = w$.
- This restriction is called the maximal set property.

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

### Maximal Set Property Diagram
- **s0**
- a+/ a+,b+/ x+ y+
- **s1**
- **s2**
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.

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_FA_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

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.
**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');
  endif;
endprocess
```

**Modified Maximal Set Property**

Slide 33

Slide 34

**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 `<s->` indicates that the transition is only taken if `s` is low.
- A clause of the form `<s+>` 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

Slide 37

Conditional Input Bursts

Slide 39

Conditional Input Bursts

Slide 38

Conditional Input Bursts

Slide 40

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).
- $I = \{x_1, \ldots, x_m\}$ 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.
- $m : V \to \{0, 1, *\}^m$ defines $m$ inputs upon entry to each state.
- $out : V \to \{0, 1\}^n$ defines $n$ outputs upon entry to each state.
- $cond : E \to \{0, 1, *\}^l$ defines needed conditional inputs.

Illegal XBM Machine

```
\begin{align*}
\text{No XBM Machine} & \\
\text{Shop PA, lazy active:} & \text{process} \\
\text{begin} & \text{guard(req_wine, '1'); -- winery calls} \\
& \text{assign(ack_wine, '1', 1, 3); -- receives wine} \\
& \text{guard(ack_patron, '0'); -- ack_patron reset} \\
& \text{assign(req_patron, '1', 1, 3); -- call patron} \\
& \text{guard(req_wine, '0'); -- req_wine reset} \\
& \text{assign(ack_wine, '0', 1, 3); -- reset ack_wine} \\
& \text{guard(ack_patron, '1'); -- wine purchased} \\
& \text{assign(req_patron, '0', 1, 3); -- reset req_patron} \\
\text{end process;}
\end{align*}
```

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 $(P, T, F, M_0)$ where $M_0$ is the initial marking.
Petri-net for Shop with Infinite Shelf Space

Presets and Postsets
- 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 $\bullet t'$) is the set of places $t$ is connected to (i.e., $\bullet t' = \{p \in P \mid (t, p) \in F\}$).
- 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 $\bullet p'$) is the set of transitions $p$ is connected to (i.e., $\bullet p' = \{t \in T \mid (p, t) \in F\}$).

Markings
- A marking, $M$, for a Petri net is a function that maps places to natural numbers (i.e., $M : P \rightarrow \mathbb{N}$).
- Markings can be added or subtracted using vector arithmetic.
- They can also be compared:
  $$M \geq M' \text{ iff } \forall p \in P . M(p) \geq M'(p)$$
- For a set of places, $A \subseteq P$, $C_A$ denotes the **characteristic marking** of $A$:
  $$C_A(p) = \begin{cases} 1 & \text{if } p \in A \\ 0 & \text{else} \end{cases}$$

Transition Firings
- A transition $t$ is **enabled** under the marking $M$ if $M \geq C_t$.
- In other words, $M(p) \geq 1$ for each $p \in \bullet t$.
- The firing transforms the marking as follows (denoted $M[t]M'$):
  $$M' = M - C_t + C_t'$$
- 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 \((\sigma = t_1, t_2, \ldots, t_n)\) 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 \([M]\).

\(k\)-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.
Liveness

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

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

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

Liveness Categories

- Different liveness categories can be determined more easily.
- In particular, a transition \( t \) for a given Petri net is said to be:
  1. \( \text{dead (L0-live)} \) if there does not exist a firing sequence in which \( t \) can be fired.
  2. \( \text{L1-live (potentially firable)} \) if there exists at least one firing sequence in which \( t \) can be fired.
  3. \( \text{L2-live} \) if \( t \) can be fired at least \( k \) times.
  4. \( \text{L3-live} \) if \( t \) can be fired infinitely often in some firing sequence.
  5. \( \text{L4-live or live} \) if \( t \) is L1-live in every marking reachable from the initial marking.
- A Petri net is \( \text{Lk-live} \) if every transition in the net is Lk-live.
Reachability Graph

- 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

\texttt{find\_RG}(Petri net $\langle P, T, F, M_0 \rangle$) {
  $M = M_0$; $T_\circ = \{ t \in T | M \geq C_t \}$; $\Phi = \{ M \}$; $\Gamma = \emptyset$;
  done = false;
  while (~ done) {
    $t = \text{select}(T_\circ)$;
    if ($T_\circ - \{ t \} \neq \emptyset$) then $\text{push}(M, T_\circ - \{ t \})$;
    $M' = M - C_t + C_t$;
    if ($M' \notin \Phi$) then {
      $\Phi = \Phi \cup \{ M' \}$; $\Gamma = \Gamma \cup \{ (M, M') \}$;
      $M = M'$; $T_\circ = \{ t \in T | M \geq C_t \}$;
    } else {
      $\Gamma = \Gamma \cup \{ (M, M') \}$;
      if (stack is not empty) then $(M, T_\circ) = \text{pop}()$;
      else done = true; $\} \} \}
  return(\Phi, \Gamma); $}

Concurrency, Conflict, and Confusion

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

Example Reachability Graph
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.
  \[ \forall p \in P : |p\bullet| = |\bullet p| = 1 \]
- 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.
  \[ \forall t, t' \in T, t \neq t' : |\bullet t \cap \bullet t'| \neq \emptyset \Rightarrow |\bullet t| = |\bullet t'| = 1 \]
- 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.
  \[ \forall p, p' \in P, p \cap p' \neq \emptyset \Rightarrow |p\bullet| = |p'\bullet| = 1 \]
- 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.

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

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

Free Choice 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 if and only if it is strongly connected and \( M_0 \) has exactly one token.

**Theorem 4.2** (Commoner, 1971) A marked graph is live and safe if and only if 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., $\bullet S \subseteq \bullet S$).
- 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.

Checking Liveness

Theorem 4.3 (Hack, 1972) A free-choice net, $N$, is live if 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.

Example Siphon and 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 if $N$ 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.
  - \( \lambda_T : T \to N \times \{+, -\} \) 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 \( a+ \to b+ \), 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.
State Graphs (SG)

- To design a circuit from an STG, must find its state graph.
- A SG is modeled by the tuple $\{ S, \delta, \lambda_S \}$.
  - $S$ 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 $(s(0), s(1), \ldots, s(n))$, where $s(i)$ is either 0 or 1, indicating value returned by $\lambda_S$.
- We use $s(i)$ interchangeably with $\lambda_S(s)(i)$.

Algorithm to Find SG

```
findSG((P, T, F, M_0, N, N_0, h_F)) {
    M = M_0; s = N_0; S = (M); \lambda_S(M) = s;
    T_e = \{ t \in T | M \subseteq \bullet \}; done = false;
    while (! done) {
        t = select(T_e);
        if (T_e - \{ t \} \neq \emptyset) then push(M, s, T_e - \{ t \});
        if ((M - \bullet) \cap \{ s \} \neq \emptyset) then return("Not safe.");
        M' = (M - t) \cup \{ s' | x, s' \in s \};
        if (\lambda_F(t) = u+) then s'(s) = 1;
        else if (\lambda_F(t) = u-) then s'(s) = 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 | M \subseteq \bullet \};
        } else {
            if (\lambda_S(M') \neq s') then return("Inconsistent.");
            if (stack is not empty) then (M, s, T_e) = pop();
            else done = true;
        }
    }
    return((S, \delta, \lambda_S));
}
```

Implied State

- If in state $s_i$, there exists a transition on signal $u_i$ to $s_j$ [i.e., $\exists (s_i, t, s_j) \in \delta : \lambda_T(t) = u_i + \land \lambda_T(t) = u_i -$], 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 $(s'(0), s'(1), \ldots, s'(n))$ of the implied values.
- The function $X : S \rightarrow 2^N$ returns the set of excited signals in a given state [i.e., $X(s) = \{ u_i | s(i) \neq s'(i) \}$].
- 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”.

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.

\[
\forall (s_i, t, s_j) \in \delta, \forall u \in N . \quad (\lambda_T(t) \neq u \Leftrightarrow s_i(u) = s_j(u)) \\
\lor \quad (\lambda_T(t) = u \Leftrightarrow s_i(u) = 0 \land s_j(u) = 1) \\
\lor \quad (\lambda_T(t) = u \Leftrightarrow s_i(u) = 1 \land s_j(u) = 0)
\]

where “\(s\)” 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.

**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., \(\forall s_i, s_j \in S, s_i \neq s_j \Rightarrow \lambda(s_i) \neq \lambda(s_j)\)].
- Some synthesis methods are restricted to STGs that produce SGs with USC.

---

**Consistent State Assignment**

![Diagram](image1)

**Unique State Code**

![Diagram](image2)
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.
- $A \subseteq N \times \{+,-,\}$ is the set of atomic actions.
- $E \subseteq A \times (\text{set of events})$.
- $R \subseteq E \times E \times (\text{set of rules})$.
- $R_O \subseteq R$ is the set of initially marked rules.
- $\# \subseteq E \times E$ is the conflict relation.
**Rules**

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

**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**

\begin{align*}
x^+ & \rightarrow [2,5] \rightarrow [z] \rightarrow [2,5] \rightarrow [7,10] \rightarrow [2,5] \rightarrow [z] \\
x^- & \rightarrow [1,3] \rightarrow [x \land \neg y] \rightarrow [1,3] \rightarrow [x \land y] \\
y^+ & \rightarrow [1,3] \\
y^- & \rightarrow [1,3] \\
z^+ & \rightarrow [1,3] \\
z^- & \rightarrow [1,3]
\end{align*}
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.
Summary

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