Asynchronous Circuit Design. Chris J. Myers

ISBNs: 0-471-41543-X (Hardback); 0-471-22414-6 (Electronic)

4

Graphical Representations

One ought, every day at least, to hear a little song, read a good poem, see a fine picture, and if it were possible, to speak a few reasonable words.

—Johann Wolfgang von Goethe

Take nothing but pictures. Leave nothing but footprints. Kill nothing but time.

—Motto of the Baltimore Grotto (caving society)

One picture is worth a thousand words.

—Fred R. Barnard

In this chapter we present several methods for representing asynchronous circuits using graphs. While for large designs hardware description languages allow a clearer specification of behavior, graphs are a useful pictorial tool for small examples. They are also the underlying data structure used by virtually all automated analysis, synthesis, and verification tools. Most graphical representation methods can be loosely categorized as either *state machine*-based or *Petri net*-based. We present various different types of each and conclude with a description of *timed event/level* (TEL) *structures*, which unify some of the key properties of both.

4.1 GRAPH BASICS

In this section we present a brief introduction to basic graph terminology used in this chapter. A *graph*, G, is composed of a finite nonempty set of *vertices*,

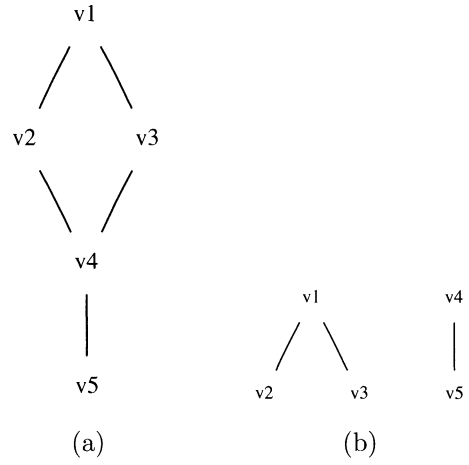


Fig. 4.1 (a) Simple undirected graph. (b) Graph which is not connected.

V, and a binary relation R on V (i.e., $R \subseteq V \times V$). A graph can be either undirected or directed.

In an *undirected graph*, R is an irreflexive and symmetric relation. By making it irreflexive, self-loops are not allowed. Since R is symmetric, for each ordered pair $(u, v) \in R$, the pair (v, u) is also in R. The set of *edges*, E, are the set of symmetric pairs in R. Each pair $\{(u, v), (v, u)\}$ is denoted (u, v) by convention. This set of edges can be empty.

Example 4.1.1 Consider the undirected graph shown in Figure 4.1(a). This graph has the following vertex set, relation, and edge set:

$$\begin{aligned} V &= \{v_1, v_2, v_3, v_4, v_5\} \\ R &= \{(v_1, v_2), (v_2, v_1), (v_1, v_3), (v_3, v_1), (v_2, v_4), \\ &\quad (v_4, v_2), (v_3, v_4), (v_4, v_3), (v_4, v_5), (v_5, v_4)\} \\ E &= \{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_3, v_4), (v_4, v_5)\} \end{aligned}$$

In a *directed graph* or *digraph*, each ordered pair of R is called a *directed edge* or *arc*. Note that as opposed to undirected graphs, R does not need to be irreflexive or symmetric. This means that if (u, v) is an arc of a digraph that (v, u) need not also be an arc. Also, self-loops, edges from a vertex to itself, are allowed.

Example 4.1.2 Consider the *digraph* shown in Figure 4.2(a). The graph has the following set of vertices and arcs:

$$\begin{aligned} V &= \{v_1, v_2, v_3, v_4, v_5\} \\ E &= \{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_3, v_4), (v_4, v_5)\} \end{aligned}$$

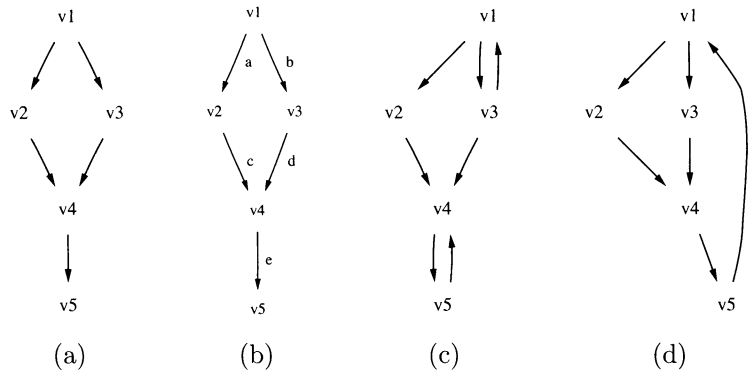


Fig. 4.2 (a) Simple digraph. (b) Labeled digraph. (c) Cyclic digraph. (d) Strongly connected digraph.

The number of elements in V (denoted $|V|$) is called the *order* of G. The number of elements in E (denoted $|E|$) is called the *size* of G. For a graph G, it is sometimes convenient to denote the vertex and edge sets as $V(G)$ and $E(G)$, respectively. If $e = (u, v) \in E(G)$, then e *joins* the vertices u and v. In an undirected graph, the edge (u, v) is *incident on* u and v. In a digraph, the arc (u, v) is *incident from* or *leaves* vertex u and is *incident to* or *enters* v. The vertex v is *adjacent* to u in a graph G if $(u, v) \in E(G)$. If $(u, v) \notin E(G)$, then u and v are *nonadjacent* vertices.

Example 4.1.3 The undirected graph in Figure 4.1(a) has order 5 and size 5. The edge (v_1, v_2) joins v_1 and v_2. The vertex v_1 is adjacent to v_2, but v_1 and v_4 are nonadjacent. In the digraph in Figure 4.2(a), the vertex v_1 is not adjacent to v_2, but v_2 is adjacent to v_1.

Let u and v be vertices of a graph G. A *u-v path* in G is an alternating sequence of vertices and edges of G, beginning with u and ending with v, such that every edge joins the vertices immediately preceding it and following it. The length of a u-v path is the number of edges in the path. If there exists a u-v path in a graph G, 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.*

Example 4.1.4 v_1, (v_1, v_2), v_2, (v_2, v_4), v_4, (v_4, v_5), v_5, (v_5, v_4), v_4, (v_4, v_3), v_3 is a v_1-v_3 path in the graph in Figure 4.1(a). We actually only need to give the vertices, since the edges are obvious. Therefore, the path can be described more concisely by v_1, v_2, v_4, v_5, v_4, v_3. This path is not simple, since it repeats vertex v_4. The graph in Figure 4.1(a) is connected while the one in Figure 4.1(b) is not since there is no path between v_4 or v_5 and the other vertices.

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 also *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 can be a cycle only if it is simple. A graph which contains no cycles is *acyclic*. An acyclic digraph is often called a *directed acyclic graph* or *DAG*.

Example 4.1.5 The undirected graph shown in Figure 4.1(a) contains the simple cycle $(v_1, v_2, v_4, v_3, v_1)$. The undirected graph shown in Figure 4.1(b) is acyclic. The digraph shown in Figure 4.2(a) is a DAG while the digraphs in Figure 4.2(c) and (d) are not. For example, the digraph in Figure 4.2(d) has the simple cycle $(v_1, v_3, v_4, v_5, v_1)$.

A digraph G is *strongly connected* if for every two distinct vertices u and v in G, there exists a u-v path and a v-u path. A graph G is *bipartite* if it is possible to partition the set of vertices into two subsets V_1 and V_2 such that every edge of G joins a vertex of V_1 with V_2.

Example 4.1.6 The digraph in Figure 4.2(d) is strongly connected while all the rest are not. The graph in Figure 4.1(a) is bipartite with $V_1 = \{v_1, v_4\}$ and $V_2 = \{v_2, v_3, v_5\}$.

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 (i.e., $L : V \rightarrow label$) or edges (i.e., $L : V \times V \rightarrow label$).

Example 4.1.7 The digraph shown in Figure 4.2(b) has a labeling function L defined as follows:

$$L = \{((v_1, v_2), a), ((v_1, v_3), b), ((v_2, v_4), c), ((v_3, v_4), d), ((v_4, v_5), e)\}$$

4.2 ASYNCHRONOUS FINITE STATE MACHINES

One of the most common graphical models for sequential logic is the *finite state machine* (FSM). The major difference between a synchronous FSM and an asynchronous FSM is when state changes are allowed. In the synchronous FSM, a clocked register is used as shown in Figure 4.3(a). In the asynchronous FSM, delay is inserted in the feedback path as shown in Figure 4.3(b). Since FSMs can be used in a similar fashion to describe both synchronous and asynchronous designs, they are a good framework for discussing sequential design somewhat independent of timing methodology.

4.2.1 Finite State Machines and Flow Tables

A FSM can be described using a 6-tuple $\langle I, O, S, S_0, \delta, \lambda \rangle$, where:

- I is the input alphabet (a finite, nonempty set of input values).
- O is the output alphabet.
- S is the finite, nonempty set of states.
- $S_0 \subseteq S$ is the set of initial (reset) states.

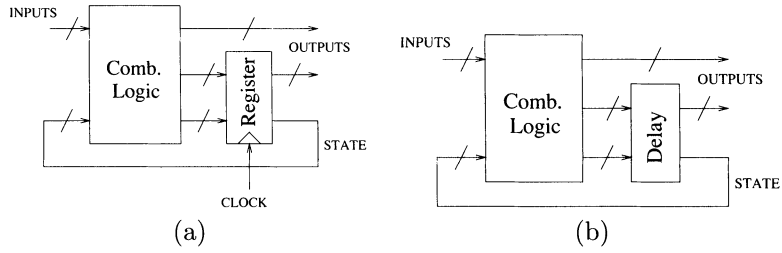


Fig. 4.3 (a) Synchronous FSM. (b) Asynchronous FSM.

- $\delta : S \times I \rightarrow S$ 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).

In a synchronous FSM, state changes can occur only on a clock edge. In an asynchronous FSM (AFSM), state changes can occur immediately after the inputs change.

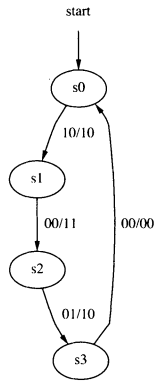
FSMs are often represented using a labeled digraph called a *state diagram*. The vertex set contains the states (i.e., $V = S$). The edge set contains the set of state transitions [i.e., $(u, v) \in E$ *iff* $\exists i \in I$ s.t. $((u, i), v) \in \delta$]. The labeling function, L, is defined by the next-state and output functions. In other words, each edge (u, v) is labeled with i/o where $i \in I$ and $o \in O$ and $((u, i), v) \in \delta$ and $((u, i), o) \in \lambda$.

Example 4.2.1 Consider the AFSM shown in Figure 4.4(a), which models the passive/active handshaking expansion for the wine shop given below.

```
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;
```

This AFSM has the following FSM characterization:

$$
\begin{aligned}
I &= \{00, 01, 10\} \\
O &= \{00, 10, 11\} \\
S &= \{s0, s1, s2, s3\} \\
S_0 &= \{s0\}
\end{aligned}
$$



(a)

req_wine / ack_patron

	00	01	11	10
s0	(s0, 00)	–	–	s1, 10
s1	s2, 11	–	–	(s1, 10)
s2	(s2, 11)	s3, 10	–	–
s3	s0, 00	(s3, 10)	–	–

ack_wine / req_patron

(b)

Fig. 4.4 (a) AFSM and (b) Huffman flow table for a passive/active wine shop (state vector *req_wine ack_patron/ack_wine req_patron*).

$$\begin{aligned}\delta &= \{((s0,10),s1),((s1,00),s2),((s2,01),s3),((s3,00),s0)\}\\ \lambda &= \{((s0,10),10),((s1,00),11),((s2,01),10),((s3,00),00)\}\end{aligned}$$

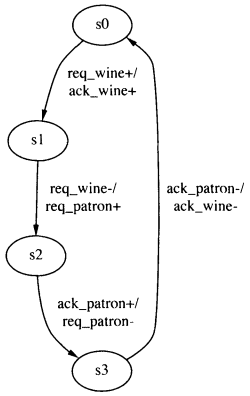
This defines a graph of the following form:

$$\begin{aligned}V &= \{s0,s1,s2,s3\}\\ E &= \{(s0,s1),(s1,s2),(s2,s3),(s3,s0)\}\\ L &= \{((s0,s1),10/10),((s1,s2),00/11),((s2,s3),01/10),\\ &\quad ((s3,s0),00/00)\}\end{aligned}$$

Each input gives the value of *req_wine* and *ack_patron*, respectively. Each output gives the value of *ack_wine* and *req_patron*, respectively. An example state transition in the AFSM would be when in state $s0$, if *req_wine* changes to '1', then *ack_wine* would be set to '1' and *req_patron* would be held at '0' as the machine makes a transition to state $s1$.

An AFSM can also be represented in a tabular form called a *Huffman flow table.* A Huffman flow table has one row for each state and one column for each input value. Each entry in the table consists of an ordered pair representing the next state and output value. When the next state is equal to the current state (i.e., the row label), it is circled to indicate that it is a *stable state.*

Example 4.2.2 Consider the Huffman flow table in Figure 4.4(b). Row $s2$ and column 00 has entry $s2, 11$ which states that the next state is also $s2$ with output 11, and it is a stable state. For row $s2$ and column 01, the entry is $s3, 10$, which indicates that the next state is $s3$ and the output should be changed to 10.



(a)

req_wine / *ack_patron*

	00	01	11	10
s0	(s0), 00	–	–	s1, 10
s1	s2, 11	–	–	(s1), 10
s2	(s2), 11	s3, 10	–	–
s3	s0, 00	(s3), 10	–	–

ack_wine / *req_patron*

(b)

Fig. 4.5 (a) BM machine and (b) Huffman flow table for a passive/active wine shop.

4.2.2 Burst-Mode State Machines

An alternative method of specifying an AFSM is using a *burst-mode* (BM) *state machine*. An example BM machine which describes the same behavior as the AFSM in Figure 4.4 is shown in Figure 4.5(a). The key difference is that in the BM machine the arcs between the states are labeled with the input and output signal transitions rather than their values. The set of input transitions is called the *input burst* and the set of output transitions is called the *output burst*. Note that the set of transitions in the input burst cannot be empty but the set of transitions in the output burst can be empty. When the set of transitions in the input burst change value, the system generates the transitions in the corresponding output burst and moves to a new state. Any BM machine can be translated into a Huffman flow table. The Huffman flow table for the BM machine in Figure 4.5(a) is shown in Figure 4.5(b). Note that, as expected, it is the same as the one from Figure 4.4(b).

> **Example 4.2.3** For the BM machine shown in Figure 4.5, in order to move from state $s0$ to $s1$, it is necessary for *req_wine* to change from '0' to '1' (indicated by *req_wine+*). Note that since *ack_patron* does not need to change value (nor is it allowed to), it is omitted from the input burst. At this point, *ack_wine* is enabled to change from '0' to '1'. Since *req_patron* does not need to change value (nor is it allowed to), it is omitted from the output burst.

Formally, a burst-mode specification is a labeled digraph which is described by a 7-tuple $\langle V, E, I, O, v_0, in, out \rangle$, where:

- 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.
- $in : V \rightarrow \{0,1\}^m$ defines the values of the m inputs upon entry to each state.
- $out : V \rightarrow \{0,1\}^n$ defines the values of the n outputs upon entry to each state.

The value of input x_i on entering state v is denoted by $in_i(v)$, and the value of output z_j on entering state v is denoted by $out_j(v)$. Intuitively, the formalism requires every state in the (unminimized) BM specification to be entered at a single *unique entry point.*

For a BM machine, two edge-labeling functions $trans_i : E \rightarrow 2^I$ and $trans_o : E \rightarrow 2^O$ can be derived to specify the transitions in the input burst and output burst, respectively. For any edge $e = (u, v) \in E$, an input x_i is in the input burst of e if it changes value between u and v [i.e., $x_i \in trans_i(e)$ *iff* $in_i(u) \neq in_i(v)$]. The function $trans_o$ can be defined in a similar manner.

A BM machine must satisfy the *maximal set property.* This property states that no input burst leaving a given state can be a subset of another leaving the same state, since the behavior in that state would then be ambiguous [i.e., $\forall (u,v),(u,w) \in E : trans_i(u,v) \subseteq trans_i(u,w) \Rightarrow v = w$].

> **Example 4.2.4** The BM machine in Figure 4.6(a) violates the maximal set property since in state $s0$ the input burst $a+$ is a subset of the input burst $a+, b+$. This means that if a changes from '0' to '1' in state $s0$ before b changes, the circuit would not know whether to move to state $s1$ or wait for b to change.

Not every *BM state diagram* represents a legal BM machine. If a BM state diagram is mislabeled with transitions that are not possible, it is impossible to define the *in* and *out* functions in a consistent fashion. These are cases where the labeling of arcs is obviously incorrect in that there must be a strict alternation of rising and falling transitions on every input and output signal, across all paths of the specification.

> **Example 4.2.5** Consider the BM machine in Figure 4.6(b), which begins with all signals low in state $s0$. In state $s0$, the transition $b-$ does not make sense since b is already low. After a changes to '1', the machine can set x to '1' and change to state $s1$. At this point, b can change to '1', causing y to be set to '1' and return to state $s0$. However, now the machine is in state $s0$ with all signals high. In this case, the transition $b-$ is okay, but the transition $a+$ does not make sense. Furthermore, the value on entry of all input and output signals is not unique. Therefore, the functions *in* and *out* cannot be defined. The BM state diagram shown in Figure 4.6(c) has the same behavior as the one in Figure 4.6(b), but it is now a legal BM machine.

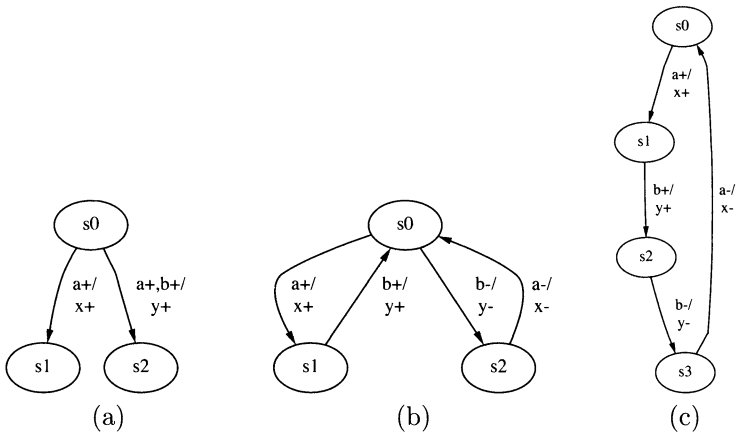


Fig. 4.6 (a) BM machine which violates maximal set property. (b) BM state diagram which is not a BM machine. (c) BM state diagram equivalent to that in (b) which is a BM machine.

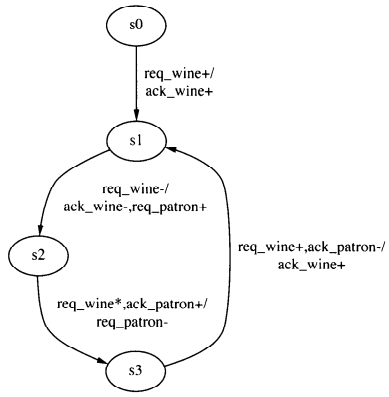
4.2.3 Extended Burst-Mode State Machines

One important limitation of AFSM and BM machines is the strict regulation that changes in signal values must follow a prescribed order: inputs change, outputs and state signals change, then state signals are fed back. 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. The idea is that some inputs in a burst may be allowed to change once monotonically along a sequence of bursts rather than having to change in a particular burst.

Example 4.2.6 The handshaking expansion shown below cannot be specified with an ordinary BM machine.

```
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;
```

The problem is *req_wine* can change from '0' to '1' at any point after *ack_wine* goes to '0'. It can, however, be specified as an XBM machine using directed don't cares as shown in Figure 4.7(a). Initially, this machine waits in state $s0$ for *req_wine* to change to '1', and it then sets



(a)

req_wine / ack_patron

	00	01	11	10
s0	(s0) 00	–	–	s1, 10
s1	s2, 01	–	–	(s1) 10
s2	(s2) 01	s3, 00	s3, 00	(s2) 01
s3	(s3) 10	(s3) 10	(s3) 10	s1, 10

ack_wine / req_patron

(b)

Fig. 4.7 (a) XBM machine and (b) Huffman flow table for a passive/active wine shop.

ack_wine to '1' and moves to state $s1$. In state $s1$, the machine waits for *req_wine* to change to '0', and it then sets *ack_wine* to '0' and *req_patron* to '1' and moves to state $s2$. Once *ack_wine* goes to '0', it is possible for *req_wine* to be set to '1' again at any time. This is indicated by putting *req_wine** in the input burst leaving $s2$. The "*" means that *req_wine* is allowed to change but is not required to change in state $s2$. In other words, the machine moves from state $s2$ to state $s3$ when *ack_patron* changes to '1' regardless of whether or not *req_wine* has been set to '1'. In state $s3$, in addition to *ack_patron* changing to '0', *req_wine* must complete (if it has not already) its transition to '1' before the machine moves to state $s1$. The Huffman flow table for the XBM machine is shown in Figure 4.7(b). Note that the effect of the directed don't care in state $s2$ is that *req_wine* can change without creating a state change. The signal *req_wine* also may not change before moving to state $s3$.

The use of directed don't cares is restricted. Before describing their restriction, it is useful to define a few terms. First, 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 in any state directly preceding the current one. Each input burst must include at least one compulsory transition.

Example 4.2.7 The XBM machine in Figure 4.8 is illegal. The transition leaving state $s0$ has no compulsory transition, since *req_wine* has a directed don't care transition from the preceding state, $s3$.

The presence of directed don't cares requires a modification of the maximal set property. In addition to the subset violation from Figure 4.6(a), one must also restrict out specifications of the form shown in Figure 4.9. In Figure 4.9(a), if b changes to '1' before a changes, it is not possible to distinguish whether the machine should move to state $s2$ or wait for a to change and

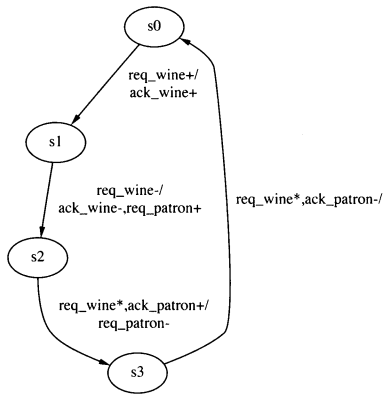


Fig. 4.8 Illegal XBM machine for a passive/active wine shop.

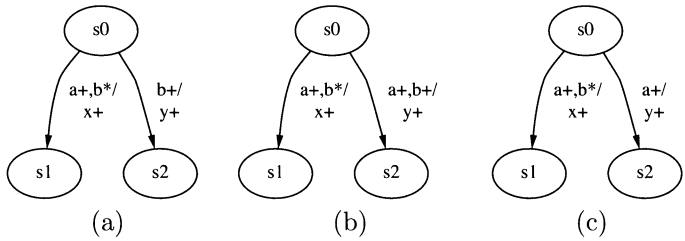


Fig. 4.9 Violations of the maximal set property in three XBM machines.

move to state $s1$. In Figure 4.9(b), after a and b have both changed to '1', the machine does not know whether to go to state $s1$ or state $s2$. In Figure 4.9(c), after a changes to '1' again, both transitions are possible, leaving the machine in a state of confusion. In general, if the compulsory transitions of one input burst leaving a state is a subset of all possible transitions in another input burst leaving the same state, then the maximal set property is violated.

Another important limitation of BM machines is their inability to express conditional behavior. It is often useful to make a decision of future behavior based on the level of a particular signal. To support this type of behavior, XBM machines allow *conditional input bursts.* A conditional input burst includes a regular input burst as defined earlier and a *conditional clause* that restricts the validity of the input burst. A clause of the form $< s- >$ indicates that the transition is taken only if s is low. A clause of the form $< s+ >$ indicates that the transition is taken only if s is high. The signal in the conditional clause must be stable before every compulsory transition in the input burst. This is a form of setup requirement, analogous to the one in a synchronous system. A specified conditional signal must be stable and valid

at some setup time before any compulsory signal arrives. The compulsory signals therefore act like a clock, to sample the conditional signal's value. The conditionals also have a hold-time requirement before they can change again. When a conditional signal is not specified, it is free to change in an arbitrary fashion.

Example 4.2.8 Let us consider the situation where a second patron moves to town. The shop has decided to sell wine of type '0' to *patron1* and type '1' to *patron2*. The VHDL model is given below. In this model, after *req_wine* goes to '0', the value of *shelf* is sampled. If *shelf* is low, the machine sets *req_patron1* to '1'. If *shelf* is high, the machine sets *req_patron2* to '1'.

```
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;
```

The XBM machine for the two-patron wine shop is shown in Figure 4.10. The two transitions leaving state $s1$ are annotated with conditional clauses on the signal *shelf*. The signal shelf must be stable a setup time before the compulsory transition which is *req_wine*– in both cases. The Huffman flow table for this XBM machine is shown in Figure 4.11. Since there are three input signals and one level signal, it gets quite complicated. Note that in all states except $s1$, the level signal, *shelf*, is free to change back and forth without changing the behavior. In state $s1$, *shelf* can change up to a setup time before *req_wine* goes low. At that point, the value of *shelf* is sampled and the next state depends on its value. A hold time after reaching the appropriate next state, *shelf* is again free to change value.

The use of conditional input bursts again changes the maximal set property. For example, in Figure 4.12(a), if we ignore the conditional, this machine violates the maximal set property. However, if s is stable before a changes to 1, the transitions are distinguishable, so there is no violation of the maximal set property. If, on the other hand, s is stable before b changes to 1, but

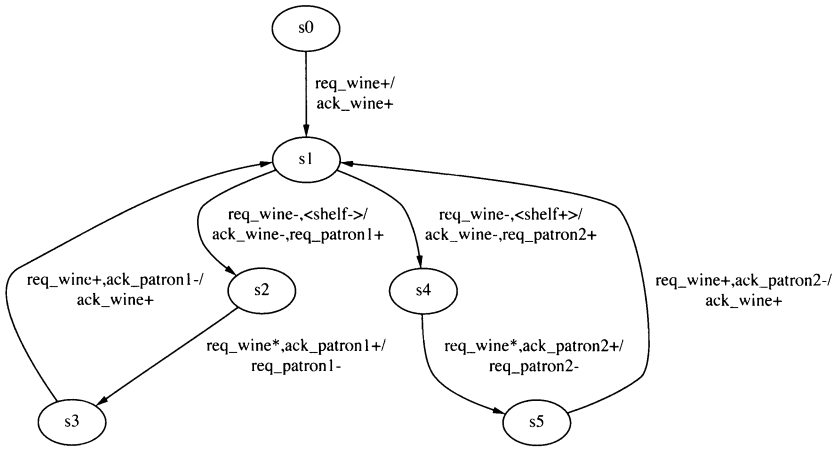


Fig. 4.10 XBM machine for wine shop with two patrons.

req_wine / ack_patron1 / ack_patron2 / shelf

	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), 010	(s4), 010	–	–	–	–	s5, 000	s5, 000
s5	(s5), 010	(s5), 010	(s5), 000	(s5), 000	–	–	–	–

ack_wine / req_patron1 / req_patron2

req_wine / ack_patron1 / ack_patron2 / shelf

	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), 010	(s4), 010
s5	–	–	–	–	(s5), 000	(s5), 000	s1, 100	s1, 100

ack_wine / req_patron1 / req_patron2

Fig. 4.11 Huffman flow table for wine shop with two patrons.

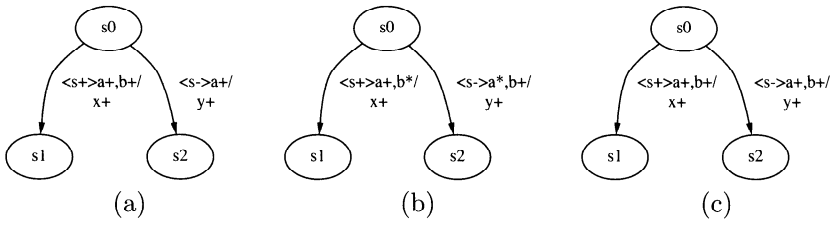


Fig. 4.12 Maximal set property and conditional input bursts in three XBM machines.

not before a changes, we would have a maximal set property violation. For example, for the sequence of transitions $a+$, $s+$, $b+$, it is not clear whether the machine ends up in state $s2$ or state $s1$. Therefore, as a conservative set up requirement, the conditional signal must be setup before any compulsory transition in any input burst leaving the state. This means that for this example we would require that s is set up before either a or b is allowed to change. In Figure 4.12(b), $a+$ is a compulsory transition for one transition and $b+$ is a compulsory transition for the other, so we again require that s is set up before either is allowed to change. In Figure 4.12(c), assume that b is a directed don't care in the state preceding state $s0$. This means that we don't know b's value when we enter state $s0$. It also means that b is not a compulsory transition for either input burst. Therefore, in this case, s must stabilize only before a can change, but b is allowed to change before s has stabilized.

XBM machines can also be described by a labeled digraph which is given formally by a 9-tuple $\langle V, E, I, O, C, v_0, in, out, cond \rangle$, where:

- 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 the values of the m inputs upon entry to each state.
- $out : V \rightarrow \{0, 1\}^n$ defines the values of the n outputs upon entry to each state.
- $cond : E \rightarrow \{0, 1, *\}^l$ defines the values of the conditional inputs needed to take a state transition.

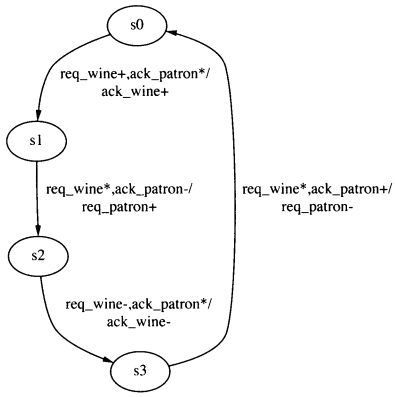


Fig. 4.13 Illegal XBM machine for reshuffled passive/lazy-active wine shop.

Again for XBM machines, the two edge-labeling functions $trans_i : E \rightarrow 2^I$ and $trans_o : E \rightarrow 2^O$ can be derived to specify the transitions in the input burst and output burst, respectively. For any edge $e = (u, v) \in E$, an input x_i is in the input burst of e if it changes or can change value between u and v [i.e., $x_i \in trans_i(e)$ *iff* $in_i(u) \neq in_i(v) \vee in_i(v) = *$]. The transition x_i+ is in the input burst *iff* $in_i(v) = 1$ and $in_i(u) \neq 1$, x_i- is in the burst *iff* $in_i(v) = 0$ and $in_i(u) \neq 0$, and x_i* is in the burst iff $in_i(v) = *$.

Although XBM machines are capable of specifying more general behavior than either AFSMs or BM machines, they are still incapable of specifying arbitrarily concurrent systems.

Example 4.2.9 Consider the following handshaking expansion and the XBM machine shown in Figure 4.13. This XBM machine appears to model the handshaking expansion. However, it is not legal since none of the input bursts have a compulsory transition.

```
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;
```

4.3 PETRI NETS

In order to specify highly concurrent systems, variants of another graphical representation method, the *Petri net*, are often used. Rather than specifying system states, these methods describe the allowed interface behavior. The interface behavior is described by means of allowed sequences of transitions, or *traces*. In this section we describe Petri nets and one of their variants, the *signal transition graph* (STG), which has been used to model asynchronous circuits and systems.

4.3.1 Ordinary Petri Nets

An *ordinary Petri net* is a bipartite digraph. The vertex set is partitioned into two disjoint subsets P, the set of *places*, and T, the set of *transitions*. Recall that in a bipartite graph, nodes from one subset must only connect with nodes from the other. In other words, the set of arcs given by the *flow relation*, 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 *marking*, M, for a Petri net is a function that maps places to natural numbers (i.e., $M : P \rightarrow N$). A Petri net is defined by a quadruple $\langle P, T, F, M_0 \rangle$, where M_0 is the *initial marking*.

Example 4.3.1 As an example, a model of a wine shop with infinite shelf space is shown in Figure 4.14(a). This Petri net is characterized as follows:

$$\begin{aligned} P &= \{p_1, p_2, p_3, p_4, p_5\} \\ T &= \{t_1, t_2, t_3, t_4\} \\ F &= \{(t_1, p_1), (p_1, t_2), (t_2, p_2), (p_2, t_1), (t_2, p_3), (p_3, t_3), (t_3, p_4), \\ &\quad (p_4, t_4), (t_4, p_5), (p_5, t_3)\} \\ M_0 &= \{(p_1, 0), (p_2, 1), (p_3, 2), (p_4, 0), (p_5, 2)\} \end{aligned}$$

The place and transition numbering is from top to bottom and left to right. A *labeled Petri net* is one which is extended with a labeling function from the places and/or the transitions to a set of labels. The Petri nets shown in Figure 4.14 have labeled transitions, and the labeling function $L : T \rightarrow \mathit{label}$ is given below.

$$L = \{(t_1, \mathit{produce}), (t_2, \mathit{receive}), (t_3, \mathit{send}), (t_4, \mathit{consume})\}$$

For a transition $t \in T$, we define the *preset* of t (denoted $\bullet t$) as the set of places connected to t [i.e., $\bullet t = \{p \in P \mid (p, t) \in F\}$]. The *postset* of 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 presets and postets of a place $p \in P$ can be similarly defined [i.e., $\bullet p = \{t \in T \mid (t, p) \in F\}$ and $p\bullet = \{t \in T \mid (p, t) \in F\}$].

Markings can be added or subtracted. They can also be compared:

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

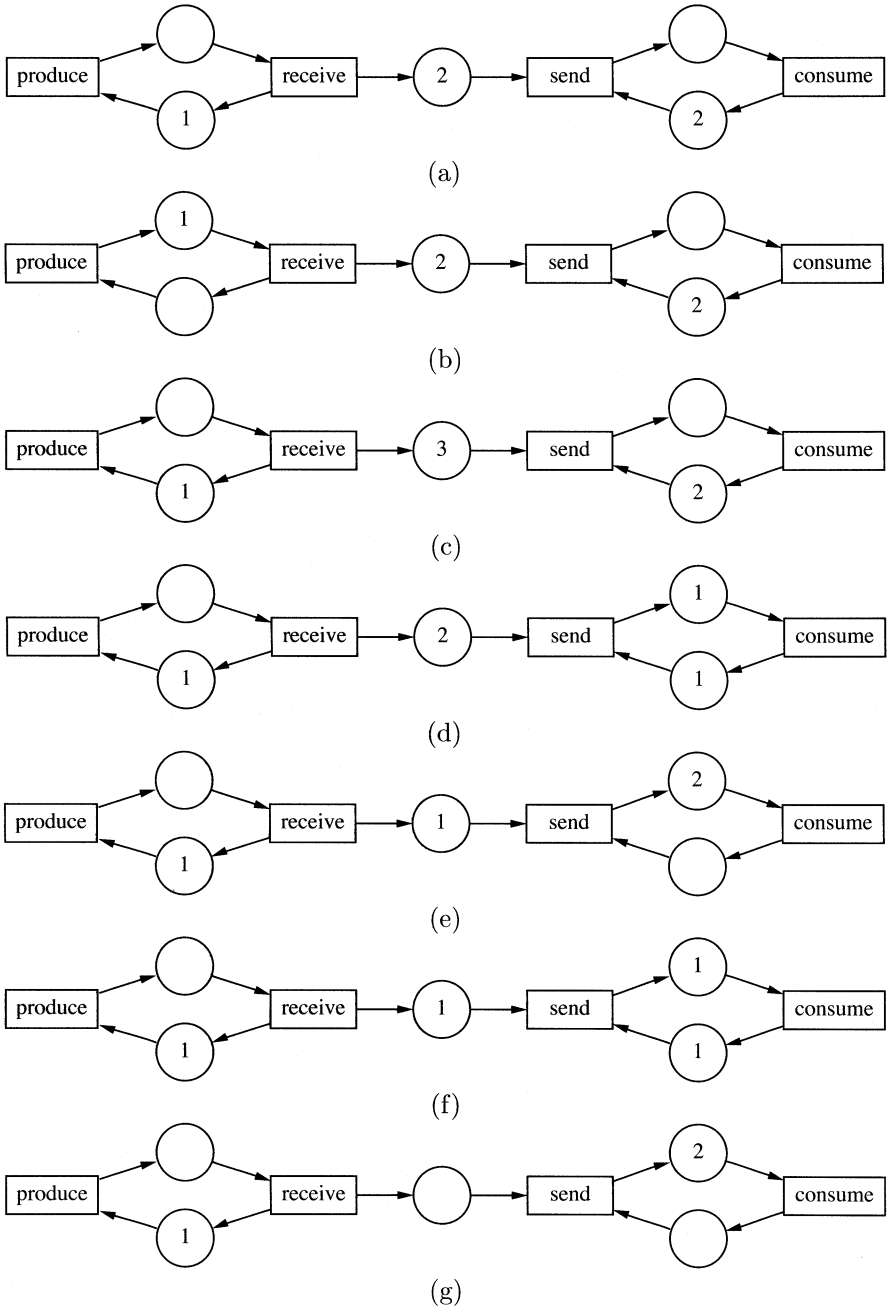


Fig. 4.14 Simple Petri net model of a shop with infinite shelf space.

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

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

The basic operation on Petri nets is the firing of a transition $t \in T$. 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 of an enabled transition transforms the marking as follows:

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

(denoted $M[t\rangle M'$). This means that when a transition t fires, a token is removed from each place in its preset (i.e., $p \in \bullet t$) and a token is added to each place in its postset (i.e., $p \in t\bullet$). The condition that a transition must be enabled before it can be fired ensures that this operation can complete without making a marking negative.

In order to understand the behavior of a system described using a Petri net, it is typically necessary to find the set of *reachable markings*. As described above, the firing of a transition transforms the marking of the Petri net into a new marking. A sequence of transition firings, or *firing sequence* (denoted $\sigma = t_1, t_2, \ldots, t_n$) produces a sequence of markings (i.e., M_0, M_1, ..., M_n). If such a firing sequence exists, we say that the marking M_n is *reachable* from M_0 by σ (denoted $M_0[\sigma\rangle M_n$). We denote the set of all markings reachable from a given marking by $[M\rangle$.

Example 4.3.2 A sample firing sequence for our Petri-net example is shown in Figure 4.14. In the initial marking shown in Figure 4.14(a), the winery is working to produce a new bottle of wine while the shop has two bottles on its shelf. For this marking, the enabled transitions are *produce* and *send*, since all places in the preset of these transitions have a marking greater or equal to 1. If the winery produces a new bottle of wine first, the marking shown in Figure 4.14(b) results. In this marking, the enabled transitions are *receive* and *send*. If the next transition is that the shop receives the bottle of wine from the winery, the new marking is shown in Figure 4.14(c). Now the shop has three bottles of wine on its shelf. In this marking, the enabled transitions are *produce* and *send*. At this point, if the shop sells a bottle of wine to the patron, the marking shown in Figure 4.14(d) results. Notice that the shop now has two bottles of wine on its shelf, the patron has one in his hand, and the patron is still capable of accepting another bottle of wine (after all, he does have two hands). Therefore, in this marking, the enabled transitions are *produce*, *send*, and *consume*. If the shop sends the patron another bottle of wine, the marking shown in Figure 4.14(e) results. In this marking, the shop's shelf still has one bottle of wine, but the patron is unable to accept it (both hands are now full). The enabled transitions are now *produce* and *consume*. The patron decides to consume one of the bottles of wine while still in the shop, leading to the marking in Figure 4.14(f). Now, the patron can go ahead and accept the last bottle of wine in the shop.

This example sequence obviously demonstrates only one of the infinite possible behaviors of this model. The number of behaviors is infinite because we could simply have a sequence of *produce* and *receive* transitions, incrementing the value of the marking of place p_3 (the shelf) each time.

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., $\forall p \in P, \forall M \in [M_0\rangle . M(p) \leq 1$]. When working with safe Petri nets, a marking can be denoted as simply a subset of places. If $M(p) = 1$, we say that $p \in M$, and if $M(p) = 0$, we say that $p \notin M$. $M(p)$ is not allowed to 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 it is 1 and without when it is 0.

Example 4.3.3 The Petri nets in Figure 4.14 are not k-bounded for any value of k. However, by adding a single place as shown in Figure 4.15(a), the net is now 2-bounded. This is accomplished by restricting the winery not to deliver a new bottle of wine unless the shop has a place to put it on its shelf. Changing the initial marking as shown in Figure 4.15(b) results in a safe Petri net. An equivalent Petri net to the one in Figure 4.15(b) using tokens is shown in Figure 4.15(c).

Another important property of Petri nets is *liveness.* A Petri net is *live* if from every reachable marking, there exists a sequence of transitions such that any transition can fire. This property can be expressed formally as follows:

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

Example 4.3.4 The Petri net shown in Figure 4.15(c) is live, but the one shown in Figure 4.15(d) is not. In this case, this shop can receive a bottle of wine, but it will not be able to send it anywhere. For this marking, the transitions *send* and *consume* are dead.

To determine if a Petri net is live, it is typically necessary to find all the reachable markings. This can be a very expensive operation for a complex Petri net. There are, however, different categories of liveness that 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.

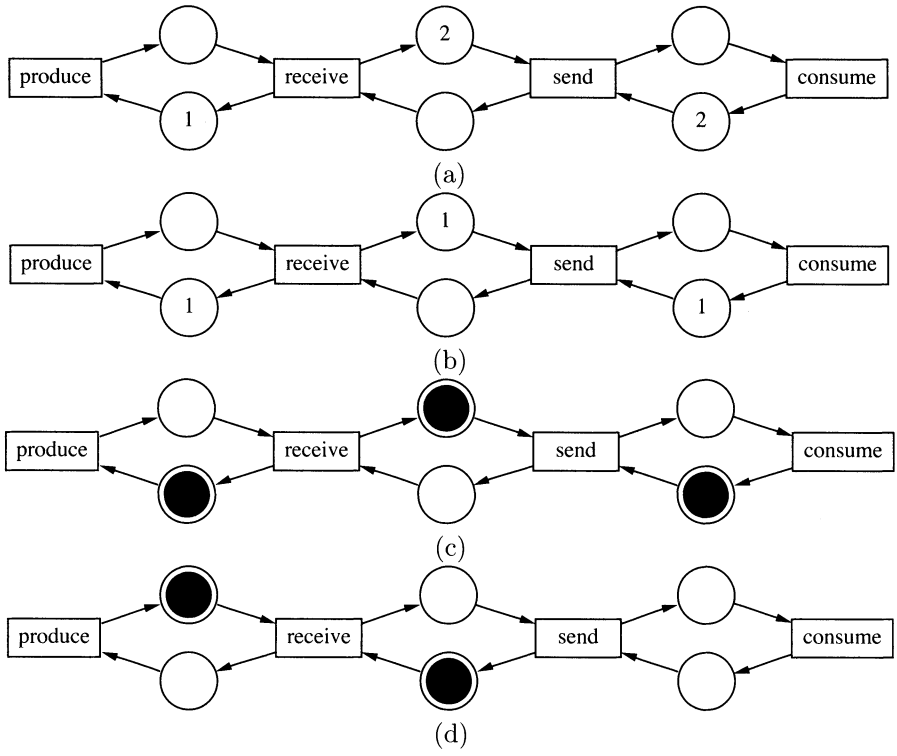


Fig. 4.15 (a) Two-bounded Petri net modeling a shop with shelf capacity of 2. (b) Safe Petri net modeling a shop with shelf capacity of 1. (c) Safe Petri net using tokens. (d) Petri net that is not live.

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

> **Example 4.3.5** The Petri net shown in Figure 4.16 describes a winery that produces wine until some point at which it chooses to deliver all its bottles to the shop. The patron must be signaled by both the winery and the shop as to when to get the wine. In this Petri net, the transition *consume* is dead, meaning that the poor patron will never get any wine. The transition *deliver* is L1-live, which means that wine will be delivered at most once. The transition *receive* is L2-live, which means that any finite number of bottles of wine can be received. Finally, the transition *produce* is L3-live, meaning that the winery could produce an infinite number of bottles of wine while delivering none of them.

When a Petri net is bounded, the number of reachable markings is finite, and they can all be found using the algorithm shown in Figure 4.17. This algorithm constructs a graph called a *reachability graph* (RG). In an RG, the vertices, Φ, are the markings and the edges, Γ, are the possible transition firings that lead the state of the Petri net between the two markings connected

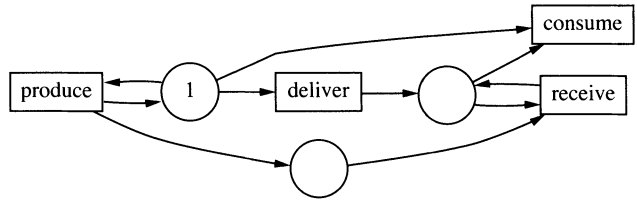


Fig. 4.16 Example in which *consume* is dead, *deliver* is L1-live, *receive* is L2-live, and *produce* is L3-live.

```
find_RG(Petri net ⟨P, T, F, M0⟩) {
  M = M0;
  Te = {t ∈ T|M ≥ C•t};
  Φ = {M};
  Γ = ∅;
  done = false;
  while (¬ done) {
    t = select(Te);
    if (Te − {t} ≠ ∅) then
      push(M, Te − {t});
    M' = M − C•t + Ct•;
    if (M' ∉ Φ) then {
      Φ = Φ ∪ {M'};
      Γ = Γ ∪ {(M, M')};
      M = M';
      Te = {t ∈ T|M ≥ C•t};
    } else {
      Γ = Γ ∪ {(M, M')};
      if (stack is not empty) then
        (M, Te) = pop();
      else
        done = true;
    }
  }
  return(Φ, Γ);
}
```

Fig. 4.17 Algorithm to find the reachability graph.

by the edge. For safe Petri nets, the vertices in this graph are labeled with the subset of the 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.

Example 4.3.6 Consider the application of the *find_RG* algorithm to the Petri net shown in Figure 4.15(c). The set M is initially set to $\{p_2, p_3, p_6\}$, and T_e is initially $\{produce, send\}$. Let us assume that

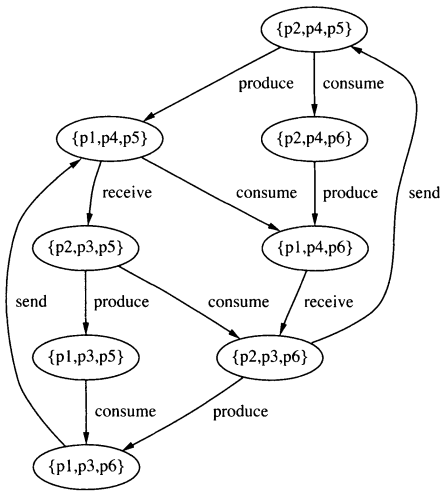


Fig. 4.18 Example reachability graph.

the *select* function chooses to fire *send*. The current marking and the remaining unfired transition, *produce*, are pushed onto the stack. The new marking is $M' = \{p_2, p_4, p_5\}$, which is not in Φ, so it is added along with a transition between these two markings to Γ. The new marking becomes the current marking, the new set of enabled transitions is $T_e = \{produce, consume\}$, and control loops back. At this point the *select* function chooses to fire *consume*, and it pushes the current marking and the remaining transition, *produce*, onto the stack. The new marking after firing *consume* is $\{p_2, p_4, p_6\}$. This marking is not found in Φ, so it is added, and a new set of enabled transitions is calculated to be $\{produce\}$. Next, the transition *produce* is fired. Since there are no remaining enabled transitions, there is nothing to push onto the stack. Firing *produce* leads to the marking $\{p_1, p_4, p_6\}$. This is again a new marking which is added, and the new set of enabled events is found to be $\{receive\}$. The *receive* transition is then fired, resulting in the marking $\{p_2, p_3, p_6\}$. This marking is found in Φ, so the marking $\{p_2, p_4, p_5\}$ and the enabled transition $\{produce\}$ are popped off the stack. This transition is fired, resulting in a new marking, $\{p_1, p_4, p_5\}$. In this new marking, the transition *receive* and *consume* are enabled. Firing *consume* results in the marking $\{p_1, p_4, p_6\}$, which is found in Φ. Therefore, we pop the marking $\{p_1, p_4, p_5\}$ and the set of enabled transition $\{receive\}$ off the stack, and we continue by firing *receive*. The rest of the markings are found using this algorithm, resulting in the RG shown in Figure 4.18.

There are several important classifications of Petri nets. Before getting into them, we first describe a few important terms. Two transitions t_1 and t_2 are said to be *concurrent* when there exists markings in which they are both

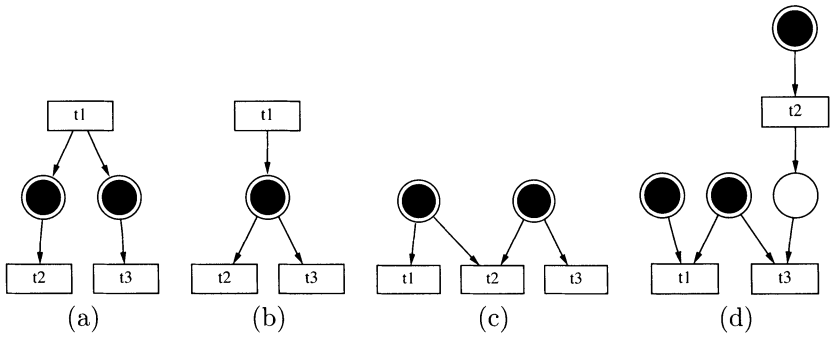


Fig. 4.19 (a) Petri net in which transitions t_2 and t_3 are concurrent. (b) Petri net in which transitions t_2 and t_3 are in conflict. (c) Petri net showing symmetric confusion. (d) Petri net showing asymmetric confusion.

enabled and they can fire in either order. Two transitions, t_1 and t_2, are said to be in *conflict* when the firing of one disables the other. In other words, t_1 can happen or t_2 can happen but not both. When concurrency and conflict are mixed, we get *confusion.*

> **Example 4.3.7** Transitions t_2 and t_3 are concurrent in Figure 4.19(a). Transitions t_2 and t_3 are in conflict in Figure 4.19(b). The Petri net in Figure 4.19(c) has *symmetric confusion* since t_1 and t_3 are concurrent, but they each conflict with t_2. The Petri net in Figure 4.19(d) has *asymmetric confusion* since t_1 and t_2 are concurrent, but t_1 conflicts with t_3 only if t_2 fires first.

The first Petri net classification is *state machines* (SM). 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 (i.e., $\forall t \in T \; . \; |\bullet t| = |t \bullet| = 1$). In other words, state machines do not allow any concurrency, but they do allow conflict.

The second classification is *marked graphs* (MG). A Petri net is a marked graph if and only if every place has exactly one transition in its preset and one transition in its postset (i.e., $\forall p \in P \; . \; |\bullet p| = |p \bullet| = 1$). Marked graphs allow concurrency, but they do not allow conflict.

The third classification is *free-choice nets* (FC). 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. More formally,

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

This can be written in an equivalent fashion on places as follows:

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

Finally, a third equivalent form is:

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

Free choice nets allow concurrency and conflict, but they do not allow confusion.

The fourth classification is *extended free-choice nets* (EFC). 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. More formally,

$$\forall p, p' \in P \; . \; p\bullet \cap p'\bullet \neq \emptyset \Rightarrow p\bullet = p'\bullet$$

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

The fifth classification is *asymmetric choice nets* (AC). 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. More formally,

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

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

Example 4.3.8 The Petri nets shown in Figure 4.15(c) is not a state machine, but it is a marked graph. The Petri net shown in Figure 4.20(a) depicts a winery that sells to two separate shops which sell to two different patrons. Also, the winery is not allowed to produce another pair of wine bottles until both patrons have received the last production. This Petri net is a state machine, but it is not a marked graph. This Petri net is also a free-choice net. If the two shops sell to the same patron and the winery is allowed to produce more wine immediately after delivering the previous batch to the shop, it is represented using the Petri net shown in Figure 4.20(b). This net is not a state machine, a marked graph, or a free-choice net. The reason this net is not free choice is that the transitions *receive1* and *receive2* share a place in their preset, but they have two places in their preset. This pair of transitions does, however, satisfy the extended free-choice requirement since the places in their preset have the same transitions in their postset. The transitions *send1* and *send2* are neither free or extended free choice. It is, however, an asymmetric choice. Consider the places in the preset of *send1*. One has only *send1* in its postset while the other has both *send1* and *send2*, so one place has a subset of the transitions in its postset compared with the other. Similarly, this is true for the places in the preset of *send2*. The net in Figure 4.20(a) is therefore an asymmetric choice net. The Petri net shown in Figure 4.20(c) depicts a model where the winery can produce two bottles of wine at once which are sold to two separate shops that sell to two separate patrons. The winery, however, has the ability to recall the wine (perhaps some bad grapes). This Petri net falls into none of these classifications. Consider the two places in the preset of transitions *send1*, *send2*, and *recall*. Both places have the transition *recall* in their postset. However, they each have a distinct transition in their postset. This net has symmetric confusion.

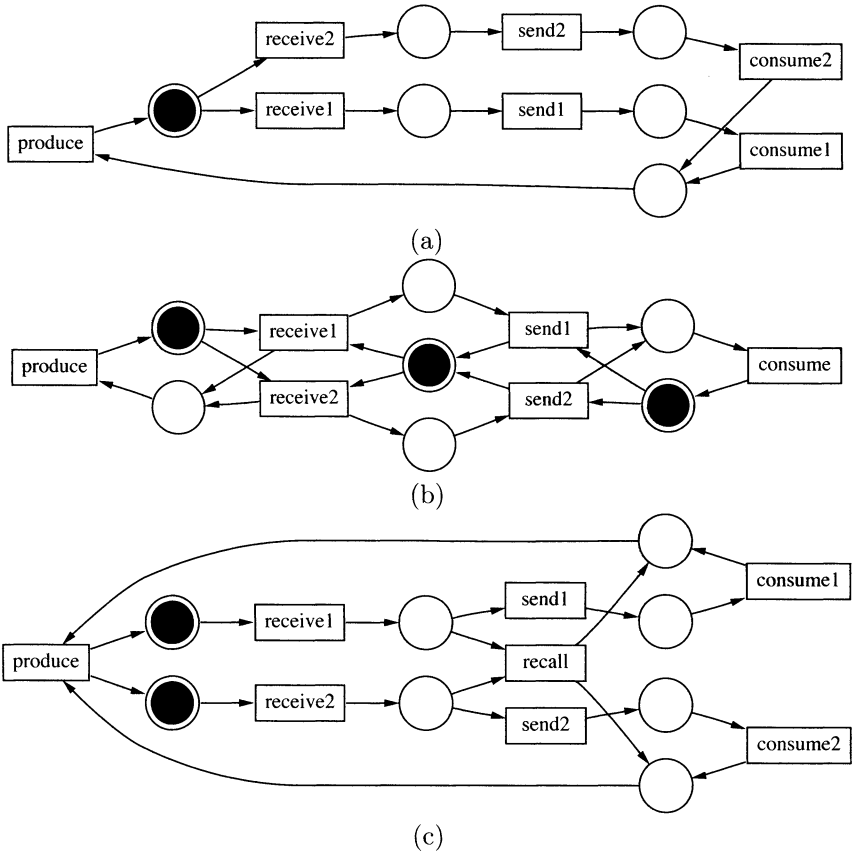


Fig. 4.20 (a) Free-choice Petri net. (b) Asymmetric choice net. (c) Petri net with symmetric confusion.

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. An example siphon is shown in Figure 4.21(a) in which the token count remains the same when firing $t1$, but reduced when $t2$ fires.

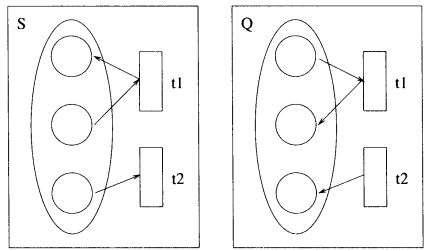


Fig. 4.21 (a) Example of a siphon. (b) Example of a trap.

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. An example trap is shown in Figure 4.21(b) in which the token count remains the same when firing $t1$, but increases when $t2$ fires.

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.*

A *state machine* (SM) *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.

Similarly, a *marked graph* (MG) *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.

The following theorems show that a live and safe free-choice net can be thought of as an interconnection of state machines or marked graphs.

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.*

Theorem 4.6 *If N is a live and safe free-choice net then N is covered by strongly connected MG-components.*

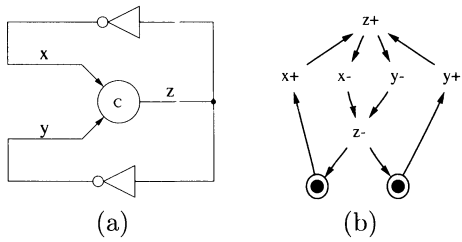


Fig. 4.22 (a) Simple C-element circuit. (b) STG model of the simple circuit.

4.3.2 Signal Transition Graphs

In order to use a Petri net to model asynchronous circuits, it is necessary to relate transitions to events on signal wires. There have been several variants of Petri nets that accomplish this, including *M-nets*, *I-nets*, and *change diagrams*. In this section we describe one of the most common variants, called a *signal transition graph* (STG). A STG is a labeled safe Petri net which is modeled with the 7-tuple $\langle P, T, F, M_0, N, s_0, \lambda_T \rangle$, 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.*

In a STG, each transition is labeled with either a rising transition, $s+$, or falling transition, $s-$. If it is labeled with $s+$, it indicates that this transition corresponds to a $0 \rightarrow 1$ transition on the signal wire s. If it is labeled with a $s-$, it indicates that the transition corresponds to a $1 \rightarrow 0$ transition on s. Note that as opposed to state machines, a STG imposes explicit restrictions on the environment's allowed behavior.

> **Example 4.3.9** An example STG model for the simple C-element circuit shown in Figure 4.22(a) is given in Figure 4.22(b). Note that implicit places (ones that have only a single transition in their preset and postset) are often omitted in STGs. In this STG, initially x and y are enabled to change from '0' to '1' (i.e., $x+$ and $y+$ are enabled). After both x and y change to '1' (in either order), z is enabled to change to '1'. This enables x and y to change from '1' to '0' (i.e., $x-$ and $y-$ are enabled). After they have both changed to '0', z can change back to '0'. This brings the system back to its initial state.

The allowable forms of STGs are often restricted to a synthesizable subset. For example, 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 arcs $a* \rightarrow b*$, there exist other arcs that ensure that $b*$ fires before

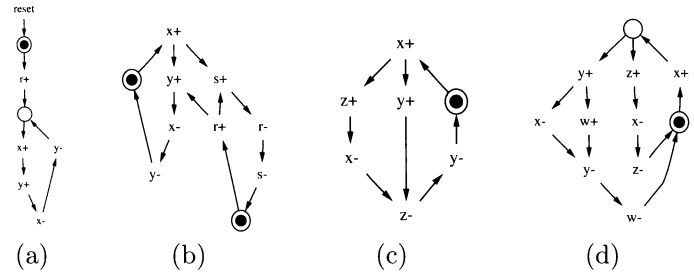


Fig. 4.23 (a) STG that is not live. (b) STG that is not safe. (c) STG that is not persistent. (d) STG that does not have single-cycle transitions.

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.

> **Example 4.3.10** The STG in Figure 4.23(a) is not live since after $r+$ fires once, it can never fire again. The STG in Figure 4.23(b) is not safe. Consider the following sequences of transitions: $x+$, $r+$, $y+$, $x-$, $y-$. At this point, if $x+$ fires before $s+$, then the arc between $x+$ and $s+$ would receive a second token. The STG in Figure 4.23(c) is not persistent. Consider the following sequence of transitions: $x+$, $z+$, $x-$. At this point, there is still a token on the arc between $x+$ and $y+$ as $x-$ is firing. The STG in Figure 4.23(d) does not have single-cycle transitions since two transitions are labeled with $x-$.

None of these restrictions is actually a necessary requirement for a circuit implementation to exist. In fact, a circuit can be built with the behavior described by each of the STGs in Figure 4.23. However, the use of these restrictions can often simplify the synthesis algorithms, as we will see later.

In order to design a circuit from an STG, it is necessary to find its *state graph* (SG). 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 binary vector $\langle s(0), s(1), \ldots, s(n)\rangle$, where each $s(i)$ is either 0 or 1, indicating the value returned by λ_S. When the context is clear, we use $s(i)$ interchangeably with $\lambda_S(s)(i)$. For each state, we can determine the value that each signal is tending to. In particular, if in a state s_i, there exists a transition on signal u_i to another reachable state s_j [i.e., $\exists(s_i, t, s_j) \in \delta\ .\ \lambda_T(t) = u_i+ \vee\ \lambda_T(t) = u_i-$], then the signal u_i is *excited.* If there does not exist such a transition, the signal u_i is in *equilibrium.* In either case, the value that 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 $\langle s'(0), s'(1), \ldots, s'(n)\rangle$ 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 $s(i) = 0$, the state in the state diagram is annotated with an "R" in the corresponding location of the state vector to indicate that the signal u_i is excited to rise. When $u_i \in X(s)$ and $s(i) = 1$, the state is annotated with an "F" to indicate that the signal is excited to fall. The algorithm to find the SG from an STG is shown in Figure 4.24.

Example 4.3.11 Consider the application of the *find_SG* algorithm to the STG in Figure 4.22. The initial state for the initial marking is $\langle 000\rangle$. In this state, $x+$ and $y+$ are enabled. Note that although $\lambda_S(s_0) = 000$, we annotate the state with the label $RR0$ to indicate that signals x and y are excited to rise. Let's assume that the algorithm chooses to fire $x+$. It then pushes the initial marking, $\langle 000\rangle$, and $y+$ onto the stack. Next, it checks if firing $x+$ results in a safety violation, which it does not. It then updates the marking and changes the state to $\langle 100\rangle$ (labeled $1R0$ in the figure). This state is new, so it adds it to the list of states, updates the state labeling function, and adds a state transition. It then determines that $y+$ is the only remaining enabled transition. Control loops back, and $y+$ is fired. There are no remaining transitions to push on the stack, so it skips to update the marking and state to $\langle 110\rangle$. This state is again new, so the state is added to the set of states. In this state, $z+$ is enabled. Continuing in this way, the rest of the SG can be found, as shown in Figure 4.25.

For a SG to be well-formed, it 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. In other words, a SG has a consistent state assignment if

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

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.

Example 4.3.12 The STG in Figure 4.26(a) does not have a consistent state assignment. If the initial state is 000, then the sequence of transitions $x+$, $y+$, $z+$, $y-$, $z-$ takes it to the same marking, which now must be labeled 100. This is not a consistent labeling, so this STG produces a SG which does not have a consistent state assignment.

A SG is said to have a *unique state assignment* (USC) if no two different states (i.e., markings) have identical values for all signals [i.e., $\forall s_i, s_j \in S, s_i \neq$

```
find_SG(⟨P, T, F, M_0, N, s_0, λ_T⟩) {
  M = M_0;
  s = s_0;
  T_e = {t ∈ T | M ⊆ •t};
  S = {M};
  λ_S(M) = s;
  done = false;
  while (¬ done) {
    t = select(T_e);
    if (T_e − {t} ≠ ∅) then
      push(M, s, T_e − {t});
    if ((M − •t) ∩ t• ≠ ∅) then
      return(''STG is not safe.'');
    M' = (M − •t) ∪ t•;
    s' = s;
    if (λ_T(t) = u+) then
      s'(u) = 1;
    else if (λ_T(t) = u−) then
      s'(u) = 0;
    if (M' ∉ S) then {
      S = S ∪ {M'};
      λ_S(M') = s'
      δ = δ ∪ {(M, t, M')};
      M = M';
      s = s';
      T_e = {t ∈ T | M ⊆ •t};
    } else {
      if (λ_S(M') ≠ s') then
        return(''Inconsistent state assignment.'');
      if (stack is not empty) then
        (M, s, T_e) = pop();
      else
        done = true;
    }
  }
  return(⟨S, δ, λ_S⟩);
}
```

Fig. 4.24 Algorithm to find a state graph.

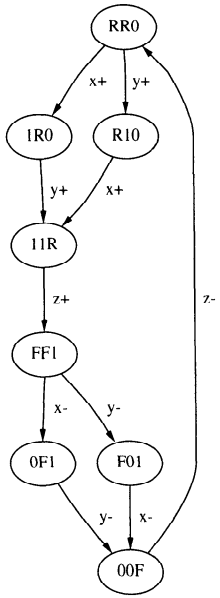


Fig. 4.25 State graph for C-element with state vector $\langle x, y, z \rangle$.

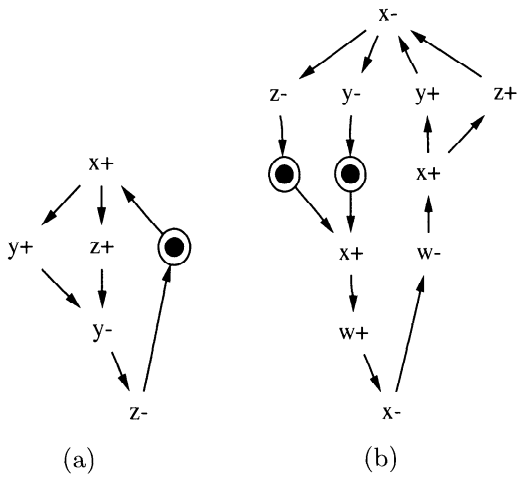


Fig. 4.26 (a) STG that does not have a consistent state assignment. (b) STG that does not have a unique state code.

$s_j \cdot \lambda(s_i) \neq \lambda(s_j)$]. Some synthesis methods are restricted to STGs that produce SGs with USC.

> **Example 4.3.13** The STG in Figure 4.26(b) produces a SG which does not have USC. In the initial marking all signals are '0' (i.e., the state vector is 0000). Consider the following sequence of transitions: $x+$, $w+$, $x-$, and $w-$. At the end of this sequence, the STG has a different marking, but all signals are again '0' (i.e., the state vector is 0000).

We conclude this section by returning to the passive/lazy-active wine shop example from the end of Section 4.2. Recall that this could not be modeled using an AFSM or even an XBM machine. This can be modeled as a STG as shown in Figure 4.27. Another example STG for the wine shop with two patrons from Section 4.2 is shown in Figure 4.28.

4.4 TIMED EVENT/LEVEL STRUCTURES

The major drawback of AFSMs is their inability to model arbitrary concurrency. The failing of Petri nets is their difficulty in expressing signal levels (see Figure 4.28). A *Timed event/level* (TEL) *structure* is a hybrid graphical representation method which is capable of modeling both arbitrary concurrency and signal levels. TEL structures support both event causality to specify sequencing and level causality to specify bit-value sampling. In this section we give an overview of TEL structures.

A TEL structure is modeled with a 6-tuple $T = \langle N, s_0, A, E, R, \# \rangle$, where:

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

The signal set, N, contains the signal wires in the specification. The state s_0 contains the initial value of each signal in N. The action set, A, contains for each signal, x, in N, a rising transition, $x+$, and a falling transition, $x-$. The set A also includes a *sequencing event*, \$, which is used to indicate an action that does not result in a signal transition. The event set, E, contains actions paired with occurrence indices (i.e., $\langle a, i \rangle$). Note that $\mathcal{N}$ represents the set of natural numbers. Rules represent causality between events. Each rule, r, is of the form $\langle e, f, l, u, b \rangle$, where:

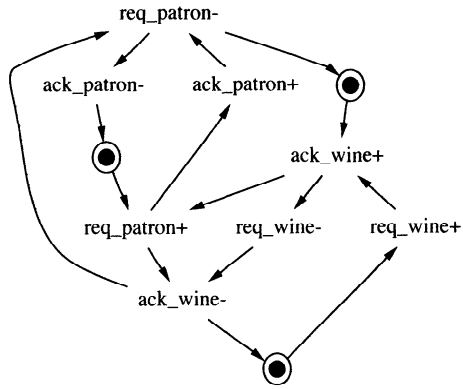


Fig. 4.27 STG for the reshuffled passive/lazy-active wine shop.

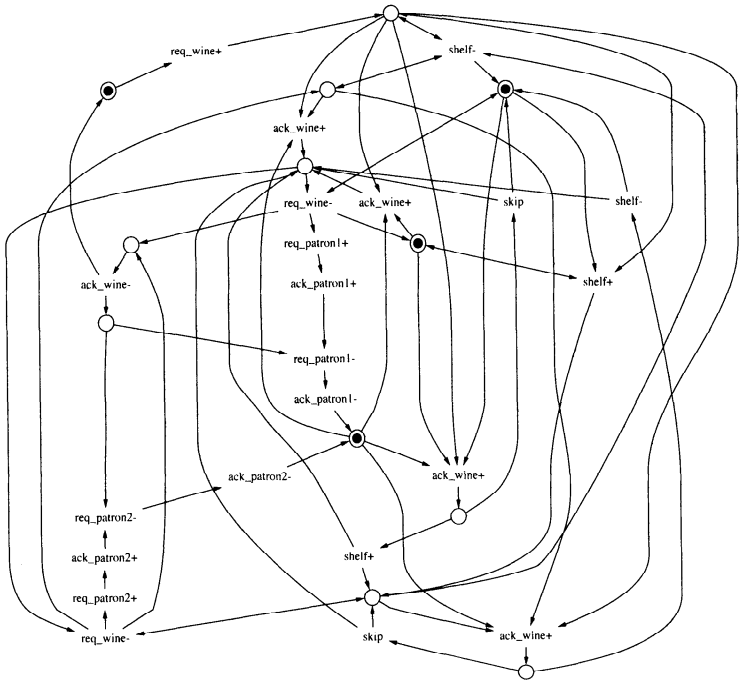


Fig. 4.28 STG for the wine shop with two patrons.

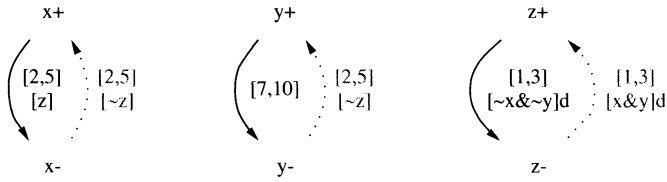


Fig. 4.29 TEL structure for a C-element and its environment.

1. e = *enabling event.*
2. f = *enabled event.*
3. $[l, u]$ = *bounded timing constraint.*
4. b = a boolean function over the signals in N.

Example 4.4.1 Figure 4.29 shows the TEL structure for a C-element and its environment. The dotted arcs represent initially marked rules, and the initial state is 000. The nodes in these graphs are events, and the arcs are rules. Each rule is annotated with a bounded timing constraint and a level expression. If the level expression is equal to **true**, it is omitted from the graph. If the rule is disabling, the level expression is terminated with a "d." Otherwise, the rule is nondisabling. There are three processes. The first represents the behavior of the signal x. The signal x rises 2 to 5 time units after z goes low, and it falls 2 to 5 time units after z rises. The behavior of y is a little different. The signal y again rises 2 to 5 time units after z goes low, but it falls 7 to 10 time units later regardless of the value of z.

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. Each rule is defined to be *disabling* or *nondisabling.* If a rule is disabling and its boolean condition becomes false after it has become enabled, the rule ceases to be enabled and must wait until the condition is true again before it can fire. If a rule is nondisabling, the fact that the boolean condition has become false is ignored, and the rule can fire as soon as its lower bound is met. For the purposes of verification, if a rule becomes disabled, this may indicate that the enabled event has a hazard which is considered a failure.

Example 4.4.2 Let us first consider the behavior of the TEL structure in Figure 4.29, ignoring timing. In the initial state, $x+$ and $y+$ are enabled. Note that $z+$ is not enabled because the expression $[x\&y]d$ is false. Let us assume that $x+$ fires first, followed by $y+$. At this point, $y-$ and $z+$ are enabled. If $y-$ fires first, the expression $[x\&y]d$ becomes

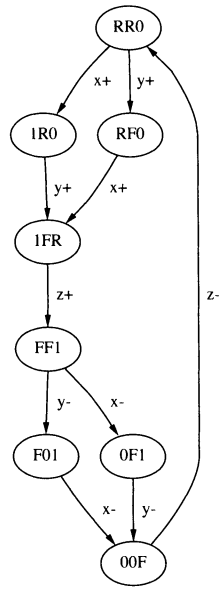


Fig. 4.30 SG for a timed C-element circuit.

disabled. This indicates that the C-element has a hazard and may glitch. Let us now consider the behavior using the timing constraints. Again in the initial state $x+$ and $y+$ are enabled. After 2 time units, the rules $\langle x-, x+, [2,5], [\sim z]\rangle$ and $\langle y-, y+, [2,5], [\sim z]\rangle$ are satisfied and can fire. By 5 time units, these rules become expired and must fire. Let's assume that $y+$ fires at time 2 (this is the worst-case scenario), which enables $y-$. After 3 more time units, $x+$ is forced to fire, enabling $z+$. After 3 more time units, we know for sure that $z+$ has fired. Therefore, it takes at most 6 time units after $y+$ fires before $z+$ fires. Since it takes at least 7 time units after y rises before y falls, we know that z does not glitch. The SG for this timed C-element circuit is shown in Figure 4.30. Note the subtle difference between this SG and the one in Figure 4.25. In this SG, $y-$ is enabled to fall sooner: namely, in states 010 (i.e., $RF0$) and 110 (i.e., $1FR$). Due to timing, however, y cannot actually fall until state 111 (i.e., $FF1$).

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. Taking the conflict relation into account, if two rules have the same enabled event and conflicting enabling events, only one of the two mutually exclusive enabling events needs to occur to cause the enabled event. This models a form of disjunctive causality. Choice is modeled when two rules have the same enabling event and conflicting enabled events. In this case, only one of the enabled events can occur.

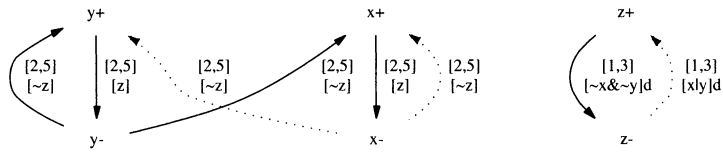


Fig. 4.31 TEL structure with conflict: namely, $x + \#y+$, $x + \#y-$, $x - \#y+$, and $x - \#y-$.

Example 4.4.3 The TEL structure shown in Figure 4.31 has the following conflicts: $x + \#y+$, $x + \#y-$, $x - \#y+$, and $x - \#y-$. In the initial state, $x+$ and $y+$ are both enabled. Even though in each case only one of its two rules is initially marked, they are still enabled since the enabling events of these rules, $x-$ and $y-$, are in conflict. In other words, only one of $x-$ or $y-$ is necessary for $x+$ or $y+$ to be enabled. Let us assume that $y+$ fires first. In the new state, $x+$ is no longer enabled since $x+$ and $y+$ are in conflict, and $z+$ is now enabled since the expression $[x|y]$ has become true. After $z+$ fires, $y-$ can fire followed by $z-$. In this state, $x+$ and $y+$ are again both enabled.

We conclude this chapter with the TEL structure for the shop process from the wine shop with two patrons from Section 4.2.3, which is shown in Figure 4.32. This TEL structure has numerous conflicts. Namely, all events in the set { *ack_wine*$-/1$, *req_patron1*$+$, *req_patron1*$-$, \$/2 } conflict with all events in the set { *ack_wine*$-/2$, *req_patron2*$+$, *req_patron2*$-$, \$/3 }. One important advantage of TEL structures is a more direct correlation with the handshaking-level specification. When you compare the STG for this example to the TEL structure, it is quite clear that the TEL structure has a more direct correspondence to the handshaking-level specification.

4.5 SOURCES

The Huffman flow table was introduced in [170, 172]. Burst mode was originally developed by Stevens [365], and it was applied successfully to a number of industrial designs by Davis's group at Hewlett-Packard [89, 99, 100, 101]. Nowick constrained and formalized burst mode into its current form, and he proposed the unique entry point and maximal set property. He also proved that any burst-mode machine satisfying these properties have a hazard-free gate-level implementation, and he developed the first hazard-free synthesis procedure [301]. Yun et al. introduced extended burst-mode machines [422].

Petri nets were introduced by Petri [312]. A couple of good surveys of Petri nets are found in [280, 311]. Seitz introduced the Petri net variant *machine nets* (M-nets) for the specification of asynchronous circuits [344]. An M-net is essentially a labeled safe Petri net in which each transition is

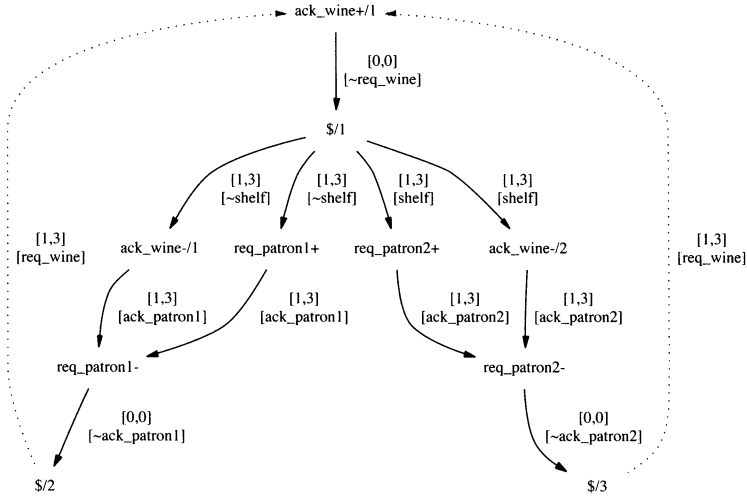


Fig. 4.32 TEL structure for the wine shop with two patrons.

labeled with either a signal name, s, or its complement, s'. If it is labeled with a signal name, it indicates that this transition corresponds to a $0 \rightarrow 1$ transition on the corresponding signal wire. If it is labeled with a signal complement, it indicates that the transition corresponds to a $1 \rightarrow 0$ transition. Molnar et al. [274] introduced another Petri net variant, *interface nets* (I-nets), for asynchronous interface specifications. An I-net is a safe Petri net with transitions labeled with interface signal names. The I-net describes the allowed interface behavior. *Change diagrams* were introduced by Varshavsky [393]. Change diagrams are quite similar to STGs, but they are capable of specifying initial behavior and a limited form of disjunctive causality. In particular, a change diagram is composed of three different types of arcs. There are *strong precedence arcs*, which model conjunctive (AND) causality. There are also *weak precedence arcs*, which model disjunctive (OR) causality. In this case, tokens only need to be present on a single incoming arc for the transition to be enabled. The last type of arcs are the *disengageable strong precedence arcs*. These arcs behave just like strong precedence arcs the first time they are encountered. They are then disengaged from the graph and are not considered in subsequent cycles. They are useful for specifying initial startup behavior. The signal transition graph (STG) was introduced by Chu [82, 83] and concurrently by Rosenblum and Yakovlev [329]. Chu also developed a method to translate AFSMs to STGs [84]. Muller and Bartky introduced the idea of a state graph [279].

TEL structures were introduced by Belluomini and Myers [34]. Timing has also been added to Petri nets by Ramchandani [320] and others.

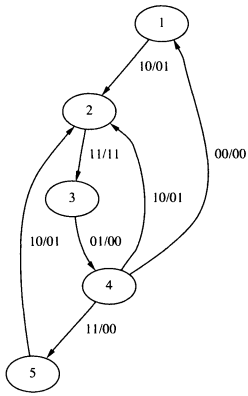


Fig. 4.33 AFSM for Problem 4.1.

Problems

4.1 Asynchronous Finite State Machines

The AFSM for a *quick-return linkage circuit* (QRL) is shown in Figure 4.33.

4.1.1. Find the flow table for the AFSM.

4.1.2. Draw a burst-mode state machine for the QRL circuit.

4.2 Burst-Mode State Machines

For each BM state diagram in Figure 4.34, determine whether:

4.2.1. It has the maximal set property.

4.2.2. It represents a BM machine.

4.3 Burst-Mode to Flow Table

Translate the BM machine shown in Figure 4.35(a) into a flow table.

4.4 Extended Burst-Mode State Machines

For the extended burst-mode state machines in Figure 4.36, determine whether they satisfy the maximal set property, and if not, explain why not.

4.5 Extended Burst-Mode to Flow Table

Translate the XBM machine shown in Figure 4.35(b) into a flow table.

4.6 Petri net Properties

For the Petri nets in Figure 4.37, determine whether they have the following properties:

4.6.1. k-bounded, and if so, for what value of k? Is it safe?

4.6.2. Live, and if not, classify each transition's degree of liveness.

4.7 Petri net Classifications

For the Petri nets in Figure 4.37, determine whether they fall into the following classifications:

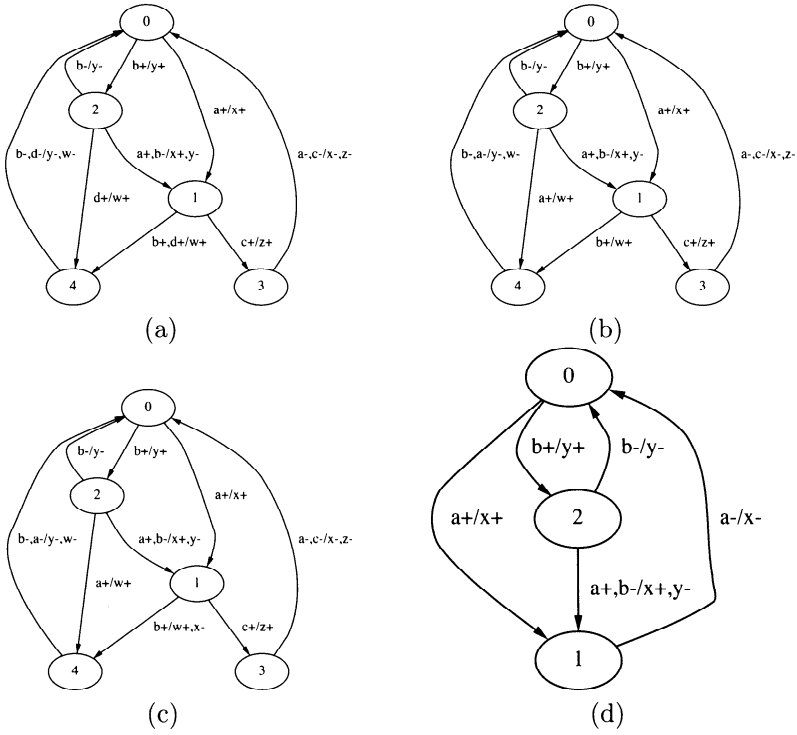


Fig. 4.34 BM machines for Problem 4.2.

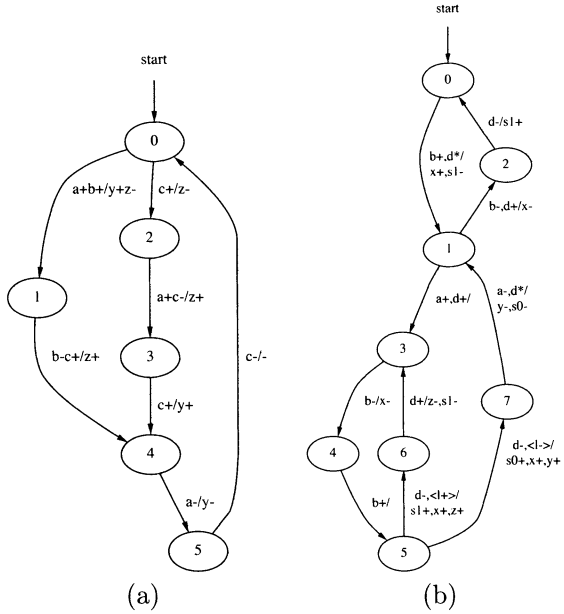


Fig. 4.35 (a) BM machine for Problem 4.3. Note that $abc = 000$ and $yz = 01$ initially. (b) XBM machine for Problem 4.5. Note that $abdl = 000-$ and $s0s1xyz = 01000$ initially.

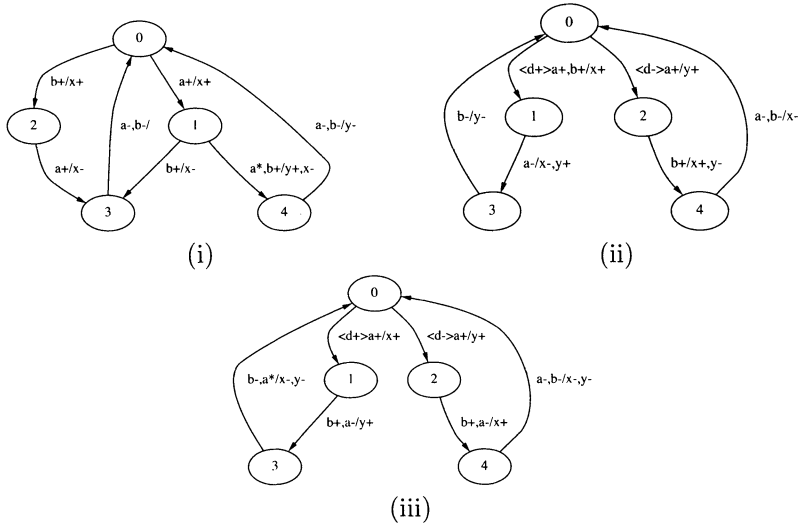


Fig. 4.36 Extended burst-mode machines for Problem 4.4.

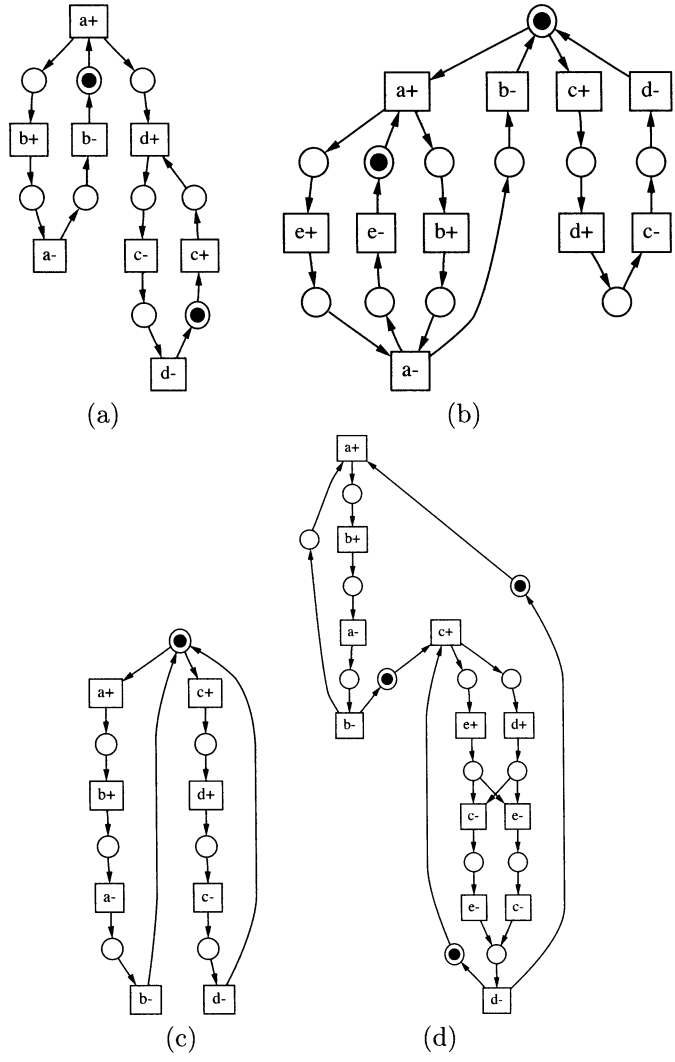


Fig. 4.37 Petri nets for Problems 4.6, 4.7, and 4.8.

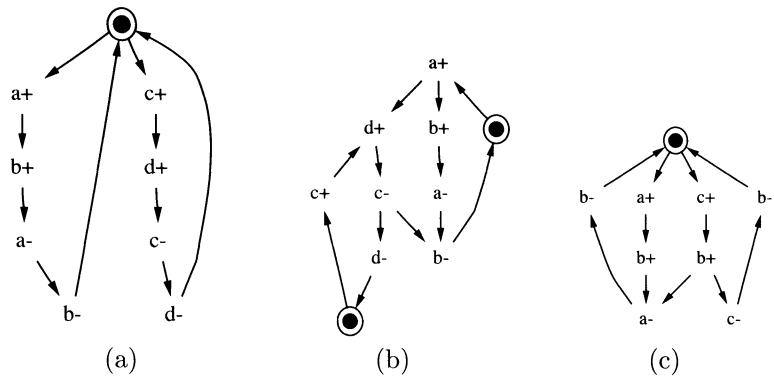


Fig. 4.38 Signal transition graphs for problems 4.9 and 4.10.

4.7.1. State machine
4.7.2. Marked graph
4.7.3. Free-choice net
4.7.4. Extended free-choice net
4.7.5. Asymmetric choice net

4.8 Reachability Graphs

For the Petri net in Figure 4.37(b), determine its reachability graph.

4.9 Signal Transition Graphs

For the signal transition graphs in Figure 4.38, state which of the following properties are satisfied or not, and if not, say why:

4.9.1. Live
4.9.2. Safe
4.9.3. Persistent
4.9.4. Single-cycle transitions

4.10 State Graphs

For the signal transition graphs in Figure 4.38, find their state graphs and determine if they have the following properties, and if not, say why:

4.10.1. Consistent state assignment
4.10.2. Unique state assignment

4.11 Signal Transition Graphs

For the signal transition graphs in Figure 4.39, state which of the following properties are satisfied or not, and if not, say why:

4.11.1. Live
4.11.2. Safe
4.11.3. Persistent
4.11.4. Single-cycle transitions

4.12 State Graphs
For the signal transition graphs in Figure 4.39, find their state graphs and determine if they have the following properties, and if not, say why:

4.12.1. Consistent state assignment

4.12.2. Unique state assignment

4.13 Signal Transition Graphs
For the signal transition graphs in Figure 4.40, state which of the following properties are satisfied or not, and if not, say why:

4.13.1. Live

4.13.2. Safe

4.13.3. Persistent

4.13.4. Single-cycle transitions

4.14 State Graphs
For the signal transition graphs in Figure 4.40, find their state graphs and determine if they have the following properties, and if not, say why:

4.14.1. Consistent state assignment

4.14.2. Unique state assignment

4.15 State Graphs
Find the state graph from the STG representation of the quick return linkage (QRL) circuit shown in Figure 4.41(a).

4.16 State Graphs
Assuming that all signals are initially low, find the SG from the STG in Figure 4.41(b).

4.17 State Graphs
Assuming that all signals are initially low, find the SG from the TEL structure in Figure 4.42. The events $a+$, $a-$, $b+$, and $b-$ all conflict with each other.

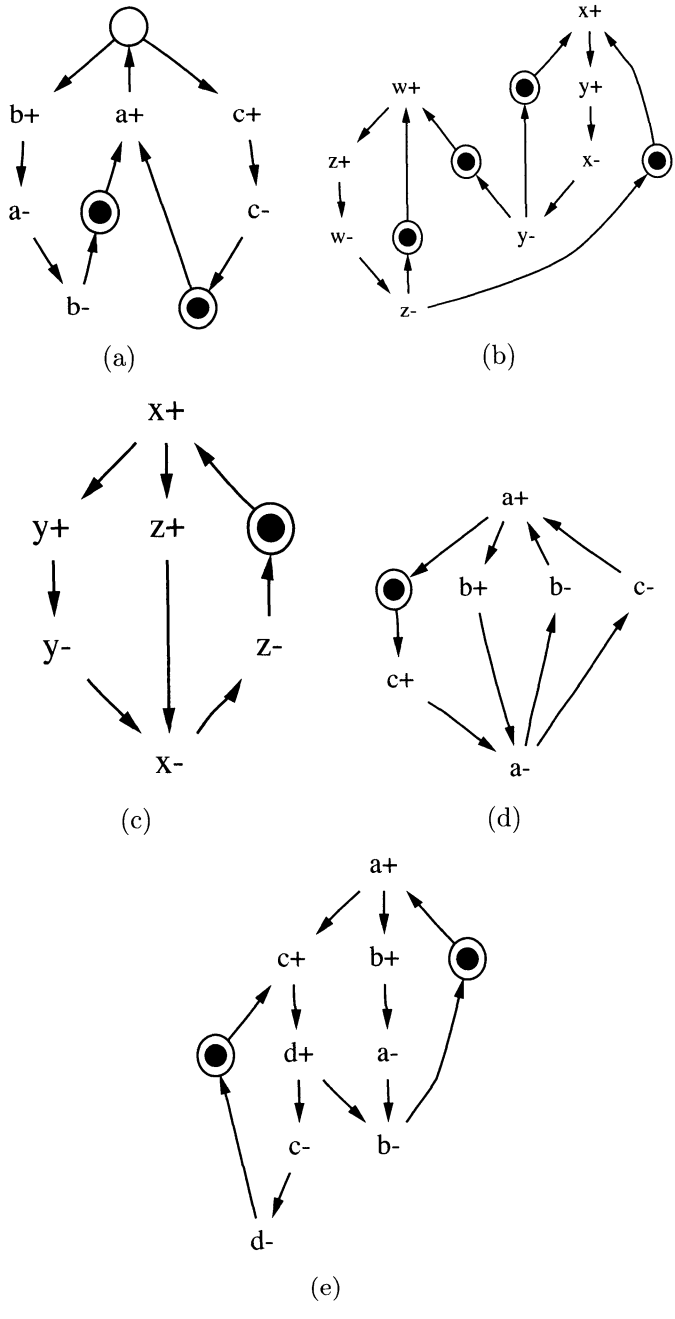


Fig. 4.39 Signal transition graphs for problems 4.11 and 4.12.

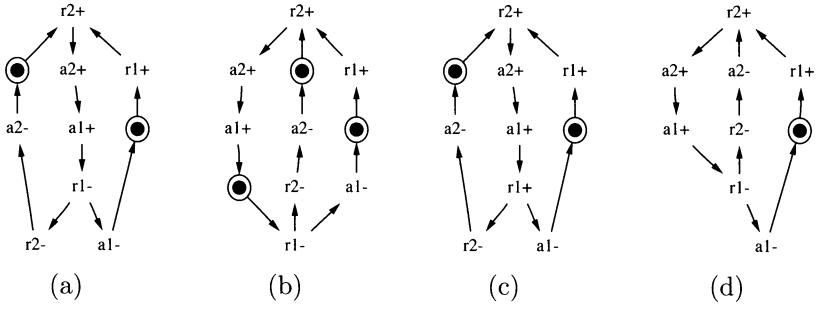


Fig. 4.40 Signal transition graphs for Problems 4.13 and 4.14.

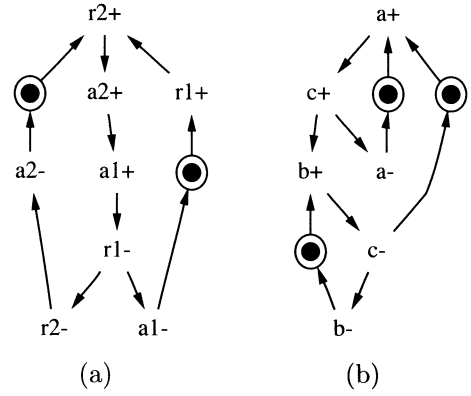


Fig. 4.41 (a) QRL circuit for Problem 4.15. (b) STG for Problem 4.16.

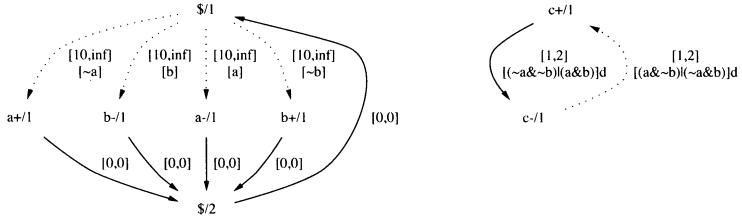


Fig. 4.42 TEL for Problem 4.17.

4.18 Translation

For the following handshaking expansion, draw graphical representations using the following methods:

- BM machine
- STG
- TEL structure

```
shopPA_dual_rail:process
begin
  guard_or(bottle0,'1',bottle1,'1');
  if bottle0 = '1' then assign(shelf0,'1',1,2);
  elsif bottle1 = '1' then assign(shelf1,'1',1,2);
  end if;
  guard(ack_patron,'1');
  assign(ack_wine,'1',1,2,shelf0,'0',1,2,shelf1,'0',1,2);
  guard_and(bottle0,'0',bottle1,'0',ack_patron,'0');
  assign(ack_wine,'0',1,2);
end process;
```