Asynchronous Circuit Design. Chris J. Myers

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

7

Timed Circuits

Time is what prevents everything from happening at once.
—John Archibald Wheeler (1911–)

Dost thou love life?Then don't squander time, for that is the stuff life is made of.
—Benjamin Franklin

We must use time as a tool, not as a couch.
—John F. Kennedy

Time is a great teacher, but unfortunately it kills all its pupils.
—Hector Berlioz

In previous chapters, synthesis of asynchronous circuits has been performed using very limited knowledge about the delays in the circuit being designed. Although this makes for very robust systems, a range of delay from 0 to infinity (as in the speed-independent case) is extremely conservative. It is quite unlikely that large functional units could respond after no delay. It is equally unlikely that gates and wires would take an infinite amount of time to respond. When timing information is known, this information can be utilized to identify portions of the state space which are unreachable. These unreachable states introduce additional don't cares in the logic synthesis problem, and they can be used to optimize the implementation that is produced. In this chapter we present a design methodology that utilizes known timing information to produce *timed circuit* implementations.

7.1 MODELING TIMING

In this section we introduce semantics to support timing information during the synthesis process. This is done using a simple example.

Example 7.1.1 Consider the case where the shopkeeper actively calls the winery when he needs wine and the patron when he has wine to sell. To save time, he decides to call the patron immediately after calling the winery, without waiting for the wine to arrive. Although the patron gets quite irate if the wine is not there, the shopkeeper simply locks the door until the wine arrives. So the process goes like this: The shopkeeper calls the winery, calls the patron, peers out the window until he sees both the wine delivery boy and the patron, lets them in, and completes the sale.

After a while, he discovers that the winery always delivers its wine between 2 and 3 minutes after being called. The patron, on the other hand, takes at least 5 minutes to arrive, even longer when he is sleeping off the previous bottle. Using this timing information, he has determined that he does not need to keep the door locked. Furthermore, he can take a short nap behind the counter, and he only needs to wake when he hears the loud voice of his devoted patron. For he knows that when the patron arrives, the wine must already have been delivered, making the arrival of the wine redundant.

The timing relationships described in the example are depicted in Figure 7.1 using a TEL structure. Recall that the vertices of the graph are events and the edges are rules. Each rule is labeled with a bounded timing constraint of the form $[l, u]$, where l is the lower bound and u is the upper bound on the firing of the rule. In this example, all events are simple sequencing events and all level expressions are assumed to be *true*.

In order to analyze timed circuits and systems, it is necessary to determine the reachable *timed states*. An *untimed state* is a set of marked rules. A timed state is an untimed state from the original machine and the current value of all the active *timers* in that state. There is a timer t_i associated with each arc in the graph. A timer is allowed to advance by any amount less than its upper bound, resulting in a new timed state.

Example 7.1.2 In the example in Figure 7.1, the rule $r_6 = \langle$*Wine is Purchased, Call Winery*$\rangle$ is initially marked (indicated with a dotted rule), so the initial untimed state is $\{r_6\}$. In the initial state, timer t_6 has value 0. Therefore, the initial timed state is $(\{r_6\}, t_6 = 0)$. In the initial state, the timer t_6 is allowed to advance by any amount less than or equal to 3.

Recall that when a timer reaches its lower bound, the rule becomes satisfied. When a timer reaches its upper bound, the rule is expired. An event enabled by a single rule must happen sometime after its rule becomes satisfied and before it becomes expired. When an event is enabled by multiple rules, it must happen after all of its rules are satisfied but before all of its rules are

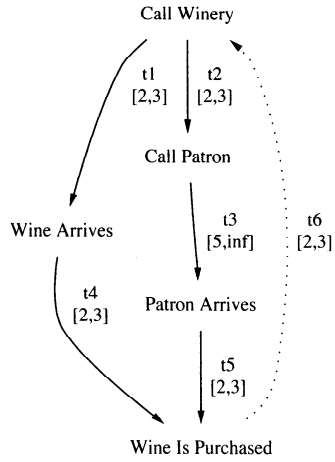


Fig. 7.1 Simple timing problem.

expired. To model the set of all possible behaviors, we extend the notion of allowed sequences to timed states and include the time at which a state transition occurs. These *timed sequences* are composed of transitions which can be either time advancement or a change in the untimed state.

Example 7.1.3 Let us consider one possible sequence of events. If $t_6 \geq 2$, the initial rule becomes satisfied. If t_6 reaches the value of 3, the rule is expired. The *Call Winery* event must happen sometime between 2 and 3 time units after the last bottle of wine is purchased. After the *Call Winery* event, timers t_1 and t_2 are initialized to 0. They must then advance in lockstep. When t_1 and t_2 reach a value of 2, the events *Wine Arrives* and *Call Patron* become *enabled.* At this point, time can continue to advance or one of these transitions can occur. Let us assume that time advances to time 2.1, and then the event *Call Patron* happens. After this event, the timer t_2 can be discarded, and a new timer t_3 is introduced with an initial value of 0. Time can be allowed to continue to advance until timer t_1 equals 3, at which point the *Wine Arrives* event will be forced to occur. Let us assume that the *Wine Arrives* when t_1 reaches 2.32. The result is that t_1 is discarded and t_4 is introduced with value 0 (note that t_3 currently has value 0.22). Time is again allowed to advance. When t_4 reaches a value of 2, the rule between *Wine Arrives* and *Wine Is Purchased* becomes satisfied, but the wine cannot be purchased because the patron has not yet arrived. Time continues until t_4 reaches a value of 3 when the same rule becomes expired, but the patron has still not arrived. At this point, we can discard t_4, as it no longer is needed to keep track of time. We replace it with a marker denoting that that rule has expired. Currently, t_3 is at a value of 3.22, so we must wait at least another 2.78 time units

before the patron will arrive. When t_3 reaches a value of 5, the rule $\langle$*Call Patron*, *Patron Arrives*, 5, inf $\rangle$ becomes satisfied, and the patron can arrive at any time. Again, we can discard the timer t_3, since there is an infinite upper bound. After the patron arrives, we introduce t_5, and between 2 and 3 time units, the wine is purchased, and we repeat. The corresponding timed sequence for this trace is as follows: (([$\{r_6\}$, $t_6 = 0$], 0), ([$\{r_6\}$, $t_6 = 2.22$], 2.22), ([$\{r_1, r_2\}$, $t_1 = t_2 = 0$], 2.22), ([$\{r_1, r_2\}$, $t_1 = t_2 = 2.1$], 4.32), ([$\{r_1, r_3\}$, $t_1 = 2.1$, $t_3 = 0$], 4.32), ([$\{r_1, r_3\}$, $t_1 = 2.32$, $t_3 = 0.22$], 4.54), ([$\{r_3, r_4\}$, $t_3 = 0.22$, $t_4 = 0$], 4.54), ([$\{r_3, r_4\}$, $t_3 = 3.22$, $t_4 = 3$], 7.54), ([$\{r_3, r_4\}$, $t_3 = 3.22$], 7.54), ([$\{r_3, r_4\}$], 9.32), ([$\{r_4, r_5\}$, $t_5 = 0$], 9.32), ([$\{r_4, r_5\}$, $t_5 = 2.2$], 11.52), ([$\{r_6\}$, $t_6 = 0$], 11.52)).

Since time can take on any real value, there is an uncountably infinite number of timed states and timed sequences. In order to perform timed state space exploration, it is necessary either to group the timed states into a finite number of equivalence classes or restrict the values that the timers can attain. In the rest of this chapter we address this problem through the development of an efficient method for *timed state space exploration.*

7.2 REGIONS

The first technique divides the timed state space for each untimed state into equivalence classes called *regions.* A region is described by the integer component of each timer and the relationship between the fractional components. As an example, consider a state with two timers t_1 and t_2, which are allowed to take any value between 0 and 5. The set of all possible equivalence classes is depicted in Figure 7.2(a). When the two timers have zero fractional components, the region is a point in space. When timer t_1 has a zero fractional component and timer t_2 has a nonzero fractional component, the region is a vertical line segment. When timer t_1 has a nonzero fractional component and t_2 has a zero fractional component, the region is a horizontal line segment. When both timers have nonzero but equal fractional components, the region is a diagonal line segment. Finally, when both timers have nonzero fractional components and one timer has a larger fractional component, the region is a triangle. Figure 7.2(a) depicts 171 distinct timed states.

Example 7.2.1 Let us consider a timed sequence from the example in Figure 7.1. Assume that the winery has just been called. This would put us in the timed state shown at the top of Figure 7.3(a). In this timed state, timers t_1 and t_2 would be initialized to 0 (i.e., $t_1 = t_2 = 0$), and their fractional components would both be equal to 0 [i.e., $f(t_1) = f(t_2) = 0$]. The geometric representation of this timed state is shown at the top of Figure 7.3(b), and it is simply a point at the origin.

In this timed state, the only possible next timed state is reached through time advancement. In other words, the fractional components

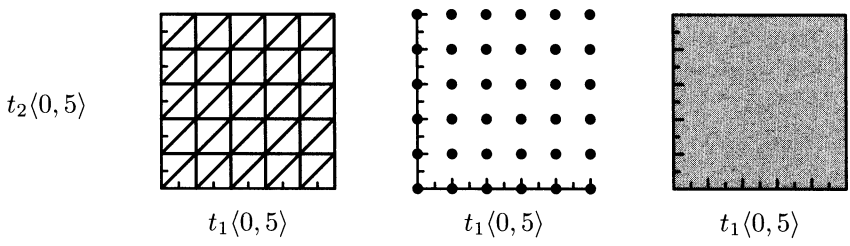


Fig. 7.2 (a) Possible timed states using regions. (b) Possible timed states using discrete time. (c) Timed state represented using a zone.

for the two timers are allowed to advance in lockstep, and they can take on any value greater than 0 and less than 1. The result is the second timed state shown in Figure 7.3(a). This timed state can be represented using a diagonal line segment as shown in the second graph in Figure 7.3(b).

Once the fractional components of these timers reach the value 1, we must move to a new timed state, increase the integer component of these timers to 1, and reset the fractional components. The result is the third timed state shown in Figure 7.3(a), which is again a point in space.

Advancing time is accomplished by allowing the fractional components to take on any value greater than 0 and less than 1, producing another diagonal line segment. When the fractional components reach 1, we again move to a new timed state where the integer components are increased to 2 and the fractional components are reset. In this timed state, there are three possible next timed states, depending on whether time is advanced, the wine arrives, or the patron is called. Let us assume that time is again advanced, leading to the timed state $t_1 = t_2 = 2, f(t_1) = f(t_2) > 0$.

In this timed state, the patron is called. In the resulting timed state, we eliminate the timer t_2 and introduce the timer t_3 with value 0. The fractional component of t_1 is greater than the fractional component of t_3, since we know that $t_1 > 0$ and $t_3 = 0$ [i.e., $f(t_1) > f(t_3) = 0$]. Pictorially, the timed state is a horizontal line segment, as shown in the seventh graph in Figure 7.3(b).

In this timed state, either the wine can arrive or time can be advanced. Let us assume that time advances. As time advances, we allow both fractional components to increase between 0 and 1. However, since we know that t_1's fractional component is greater than t_3's, the resulting region in space is the triangle shown in the eighth graph in Figure 7.3(b).

Once the fractional component for t_1 reaches 1, we must increase the integer component for timer t_1 to 3. Note that in this timed state, the fractional component can be anywhere between 0 and 1. Also note

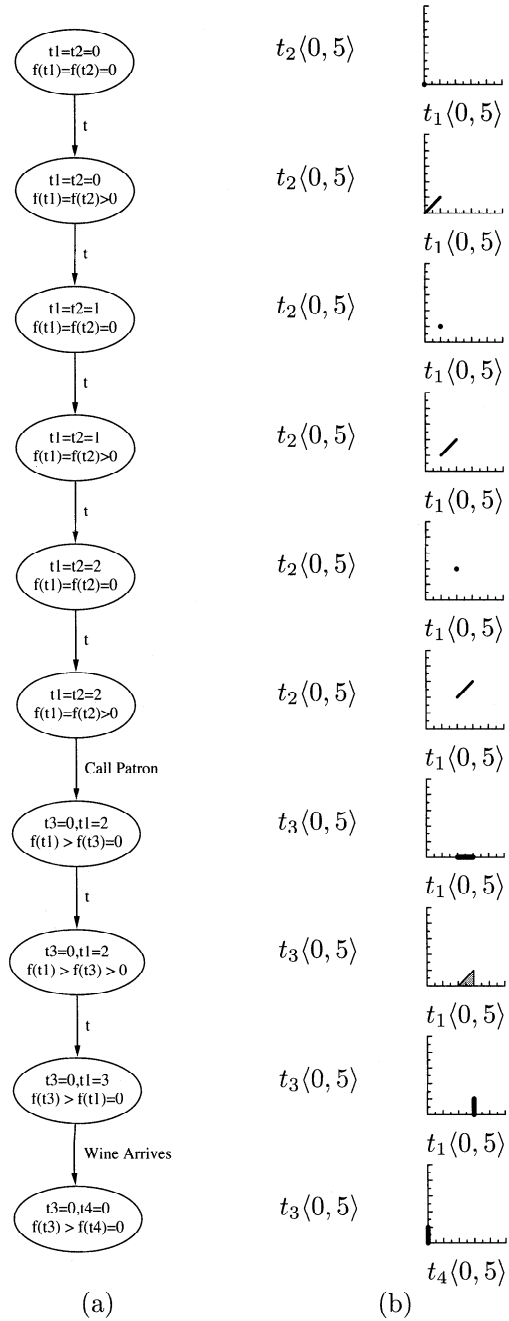


Fig. 7.3 (a) Sequence of timed states using regions. (b) Geometric representation of the sequence of timed states.

that since we reset the fractional component for t_1 to 0, the fractional component for t_3 is now greater than that for t_1.

In this timed state, the timer t_1 has reached its upper bound, so the event *Wine Arrives* must occur. To produce a new timed state, we eliminate the timer for t_1 and introduce the timer for t_4. We know that the fractional component for t_3 is greater than that of t_4 since $f(t_3) > 0$ and $f(t_4) = 0$.

This timed sequence represents only one of the many possible sequences of timed states from the point that the winery is called until both the patron is called and the wine arrives. The portion of the timed state space represented using regions for all possible timed sequences involving these three events is shown in Figure 7.4. Using this region-based technique, it requires 26 timed states to represent all the timing relationships for only four untimed states.

Indeed, the timed state explosion can be quite severe since the worst-case complexity is

$$|S|\frac{n!}{\ln 2}\left(\frac{k}{\ln 2}\right)^n 4^{1/k}$$

where S is the number of untimed states, n is the number of rules that can be enabled concurrently, and k is the maximum value of any timing constraint.

7.3 DISCRETE TIME

For timed Petri nets and TEL structures, all timing constraints are of the form $\leq$ or $\geq$, since timing bounds are inclusive. In other words, we never need to check if a timer is strictly less than or greater than a bound. It has been shown that in this case, the fractional components are not necessary. Therefore, we only need to keep track of the *discrete-time* states. If we consider again two timers that can take values from 0 to 5, the possible timed states are shown in Figure 7.2(b). The number of timed states in this figure is now only 36. The worst-case complexity is now

$$|S|(k+1)^n$$

This represents a reduction in worst-case complexity compared to region-based techniques by a factor of more than $n!$.

Example 7.3.1 Using discrete time, each time advancement step simply increments all timers by 1 time unit. For example, after the winery has been called, we are in a state with timers t_1 and t_2 initially 0 as shown at the top of Figure 7.5. In this timed state, time is advanced by 1 time unit, bringing us to a new timed state where the timers are each equal to 1. In this timed state, time must again be advanced bringing us to a new timed state where the timers equal 2. In this timed state, time can again advance, the wine can arrive, or the patron can be called. Exploring each possibility, we derive 13 possible timed states from the

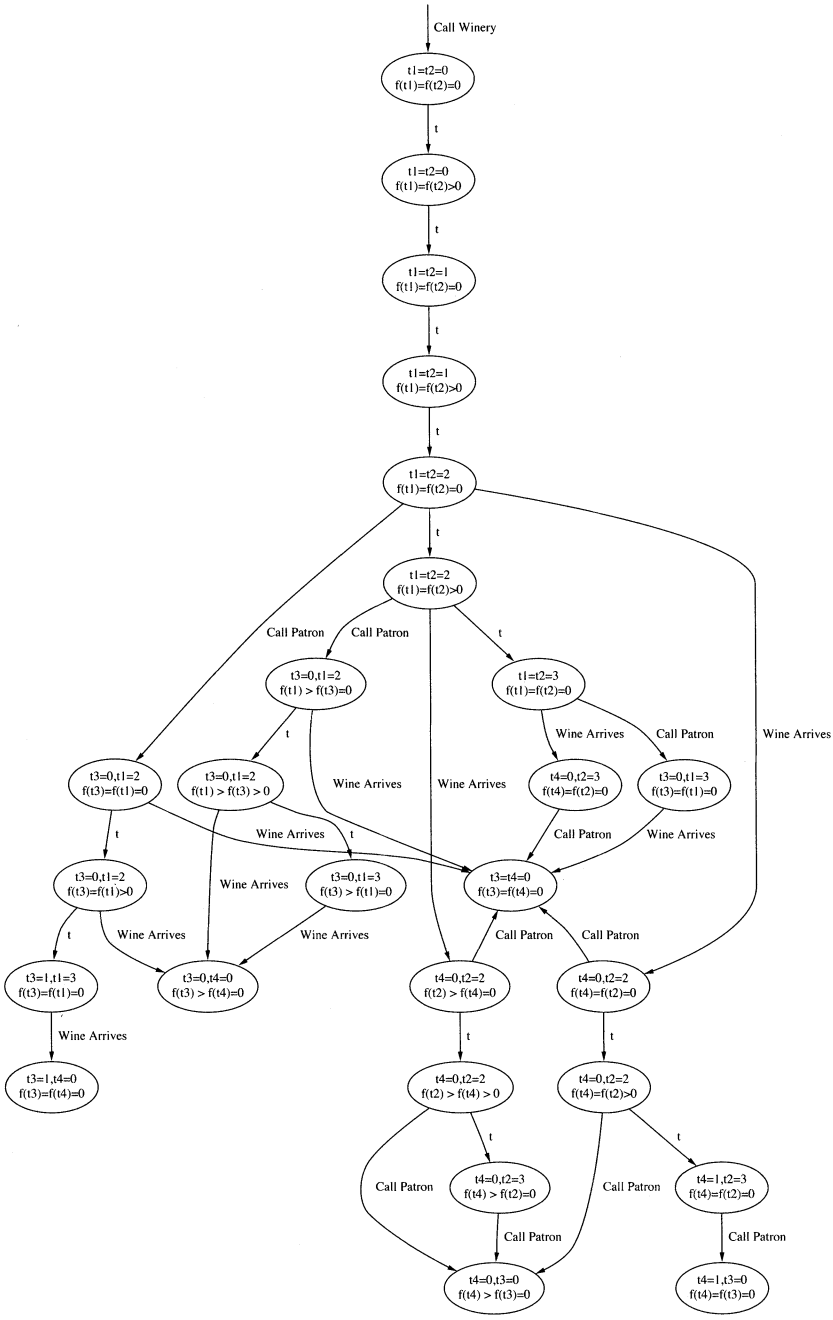


Fig. 7.4 Part of the timed state space using regions.

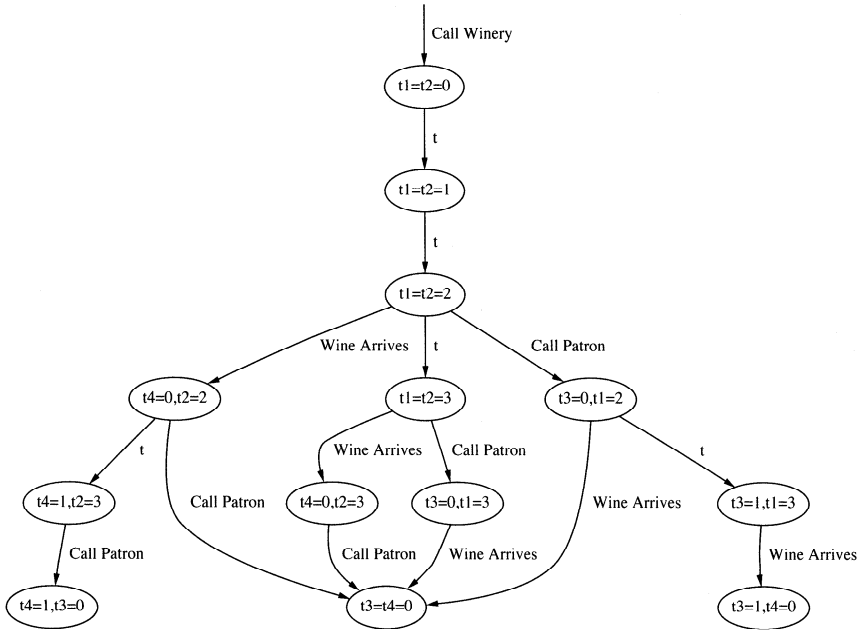


Fig. 7.5 Part of the timed state space using discrete time.

time the winery has been called until both the patron has been called and the wine has arrived. This is a reduction in half as compared with the 26 found using regions.

Unfortunately, the discrete-time technique is still exponential in the number of concurrent timers and size of the timing bounds. For example, if we change each timing bound of $[2, 3]$ to $[19, 31]$ and change $[5, \text{inf}]$ to $[53, \text{inf}]$, the total number of timed states grows from 69 to more than 3000. Changing the bounds to $[191, 311]$ and $[531, \text{inf}]$ increases the number of discrete timed states to over 300,000!

7.4 ZONES

Another approach is to use convex polygons, called *zones*, to represent equivalence classes of timed states. For example, if there are two concurrent timers that can take on any value between 0 and 5, it can be represented using a single zone as shown in Figure 7.2(c). In other words, one zone is representing 171 regions or 36 discrete-time states.

Any convex polygon can be represented using a set of linear inequalities. To make them uniform, we introduce a *dummy timer* t_0 which always takes the value 0. For each pair of timers, we introduce an inequality of the form

$$t_j - t_i \leq m_{ij}$$

This set of inequalities is typically collected into a data structure called a *difference bound matrix* (DBM).

Example 7.4.1 For the example in Figure 7.2(c), the set of linear inequalities and the corresponding DBM are shown below.

$$\begin{array}{rcl} t_0 - t_0 & \leq & 0 \\ t_1 - t_0 & \leq & 5 \\ t_2 - t_0 & \leq & 5 \\ t_0 - t_1 & \leq & 0 \\ t_1 - t_1 & \leq & 0 \\ t_2 - t_1 & \leq & 5 \\ t_0 - t_2 & \leq & 0 \\ t_1 - t_2 & \leq & 5 \\ t_2 - t_2 & \leq & 0 \end{array}$$

	t_0	t_1	t_2
t_0	0	5	5
t_1	0	0	5
t_2	0	5	0

The same zone can be represented by many different sets of linear inequalities and thus many different DBMs. During state space exploration, it is necessary to determine when you have encountered a timed state that you have seen before. In other words, when you find an untimed state that you have seen before, you must also check that the zone is the same as before. If there are multiple different representations, you may encounter the same timed state and not know it. Fortunately, there is a canonical DBM representation for each zone when all entries are maximally tight.

Example 7.4.2 Consider the following DBM:

	t_0	t_1	t_2
t_0	0	5	5
t_1	0	0	7
t_2	0	5	0

This DBM indicates the following relationship:

$$t_2 - t_1 \leq 7$$

However, the DBM also indicates these relationships:

$$t_2 - t_0 \leq 5$$

$$t_0 - t_1 \leq 0$$

If we add them together, we get the following relationship:

$$t_2 - t_1 \leq 5$$

Therefore, the first inequality is not maximally tight in that it is not possible for $t_2 - t_1$ actually to take any value in the range greater than 5 and less than 7. Therefore, the constraint can be tightened to produce our original DBM.

```
recanonicalization(M) {
  for k = 1 to n
    for i = 1 to n
      for j = 1 to n
        if (m_ij > m_ik + m_kj) then
          m_ij = m_ik + m_kj;
}
```

Fig. 7.6 Floyd's all-pairs shortest-path algorithm.

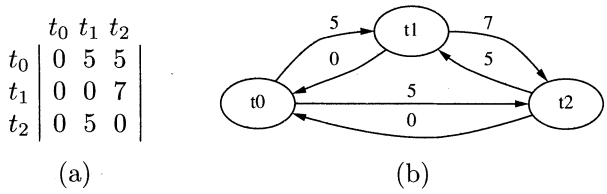


Fig. 7.7 (a) Example DBM. (b) Its digraph representation.

Finding the canonical DBM is equivalent to finding all pairs of shortest-paths in a graph. To cast it as the shortest path problem, we create a labeled digraph where there is a vertex for each timer t_i and an arc from t_i to t_j for each linear inequality of the form $t_j - t_i \leq m_{ij}$ when $i \neq j$. Each arc is labeled by m_{ij}. We use Floyd's all-pairs shortest-path algorithm to perform *recanonicalization.* Floyd's algorithm is an efficient algorithm for finding all pairs of shortest paths in a graph, and it is shown in Figure 7.6. This algorithm examines each pair of vertices t_i and t_j in turn and attempts to find an alternative route between them which passes through some t_k. If this alternative route is shorter than the direct route, we can reduce the distance of the direct route to this value without changing the shortest path. This is equivalent in the DBM case of tightening a loose bound.

Example 7.4.3 The digraph representation of the DBM shown in Figure 7.7(a) appears in Figure 7.7(b). For this graph, the algorithm would discover that the route from t_1 to t_2 through t_0 is only distance 5, while the route from t_1 directly to t_2 is distance 7. Therefore, the arc between t_1 and t_2 can be reduced to 5.

During timed state space exploration using zones, a state transition occurs as the result of a rule firing. A rule, r_j, can fire if its timer, t_j, can reach a value that makes it satisfied (i.e., $m_{0j} \geq l_j$). If this rule firing is the last one enabling an event, an event happens, leading to new rules being enabled. Whether or not an event happens, it is necessary to calculate a new zone from the preceding one. This is accomplished using the algorithm shown in Figure 7.8. This algorithm takes the DBM for the original zone, the rule that

```
update_zone(M, r_j, event_fired, R_en, R_new) {
  if m_j0 > -l_j then m_j0 = -l_j;          /* Restrict to lower bound.*/
  recanonicalize(M);                          /* Tighten loose bounds.*/
  project(M, r_j);          /* Project out timer for rule that fired.*/
  if (event_fired) then
    foreach r_i ∈ R_new {                  /* Extend DBM with new timers.*/
      m_i0 = m_0i = 0;
      foreach r_k ∈ R_new
        m_ik = m_ki = 0;
      foreach r_k ∈ (R_en - R_new){
        m_ik = m_0k;
        m_ki = m_k0;
      }
    }
  foreach r_i ∈ R_en                                  /* Advance time.*/
    m_0i = u_i
  recanonicalize(M);                        /* Tighten loose bounds.*/
  normalize(M, R_en);                  /* Adjust infinite upper bounds.*/
}
```

Fig. 7.8 Algorithm to update the zone.

fired, a flag indicating whether an event fired, the set of currently enabled rules, and the set of those rules which are newly enabled by this rule firing.

As an example, consider the firing of a rule $r_j = \langle e_j, f_j, l_j, u_j \rangle$, where e_j is the enabling event, f_j is the enabled event, l_j is the lower bound of the corresponding timer t_j, and u_j is the upper bound on the timer. The first step is to *restrict* the DBM to indicate that for this rule to have fired, its timer must have reached its lower bound. In other words, if $t_0 - t_j > -l_j$, we must set m_{j0} to $-l_j$. This additional constraint may result in some of the other constraints no longer being maximally tight. Therefore, it is necessary to recanonicalize the DBM using Floyd's algorithm. The next step is to *project* the row and column corresponding to timer t_j since once this rule has fired, we no longer need to maintain information about its timer.

If the rule firing causes an event, this event may enable new rules. For each of these newly enabled rules, we must introduce a new timer which corresponds to a new row and column in the DBM. For each newly enabled rule (i.e., $r_i \in R_{new}$), we add a new timer t_i. We initialize the lower and upper bounds of this timer (i.e., m_{i0} and m_{0i}) to 0 to represent that the timer is initialized to 0. The time separation between the timer for each pair of rules that has been newly enabled is also set to 0, as these timers got initialized at the same time. Finally, the remaining new row entries, m_{ik}, are set equal to the upper bounds of their timers t_k (i.e., m_{0k}), and the remaining new column entries, m_{ki}, are set equal to the lower bounds of their timers t_k (i.e., m_{k0}).

```
normalize(M, R_en) {
  foreach r_i ∈ R_en
    if (m_i0 < −premax(r_i)) then  /* Reduce timer to premax value.*/
      foreach r_j ∈ R_en {
        m_ij = m_ij − (m_i0 + premax(r_i));
        m_ji = m_ji + (m_i0 + premax(r_i));
      }
  foreach r_i ∈ R_en                          /* Adjust maximums.*/
    if (m_0i > premax(r_i)) then
      m_0i = max_j(min(m_0j, premax(r_j)) − m_ij);
  recanonicalize(M);                    /* Tighten loose bounds.*/
}
```

Fig. 7.9 Normalization algorithm.

The next step is to *advance time* by setting all timers to their upper bound (i.e., $m_{0i} = u_i$). The resulting DBM may now again contain entries that are not maximally tight. Therefore, we again recanonicalize the DBM.

The final step is to *normalize* the DBM to account for rules with infinite upper bounds. This is necessary to keep the state space finite. The algorithm for normalization is shown in Figure 7.9. This algorithm uses the function *premax*, which takes a rule r_i and returns u_i if it is finite and returns l_i if u_i is infinite. This algorithm works on the assumption that the value of a timer of a rule with an infinite upper bound becomes irrelevant once it has exceeded its lower bound. It simply must remember that it reached its lower bound. The algorithm adjusts timers accordingly, in three steps. First, if the lower bound of the timer on a rule r_i has already exceeded its *premax* value, the zone is adjusted to reduce it back to its *premax* value. This may reduce some timers below their *premax* or current maximum value, so it may be necessary to allow the timers to take a value exceeding *premax*. Therefore, for each timer that has exceeded its *premax* value, we find the minimum maximum value needed which does not constrain any other rules. Finally, the DBM is recanonicalized again.

Example 7.4.4 Let us illustrate timed state space exploration with zones using the example shown in Figure 7.1. The initial state has only one timer t_6, so our initial DBM is only two by two (i.e., t_0 and t_6). We begin by initializing the DBM to all zeros to represent that all timers are initially zero. We then advance time by setting the top row entry for t_6 to the maximum value allowed for t_6, which is 3. We then recanonicalize and normalize, which simply results in the same DBM. This sequence of steps is shown in Figure 7.10.

The only possible rule that can fire in this initial zone is r_6. Since the lower bound on this rule is 2, we need to restrict the timer t_6 in the DBM such that it cannot be less than 2 (see Figure 7.11). Again, recanonicalization has no effect. Next, we project out the row and column associated with timer t_6. The firing of this rule results in the

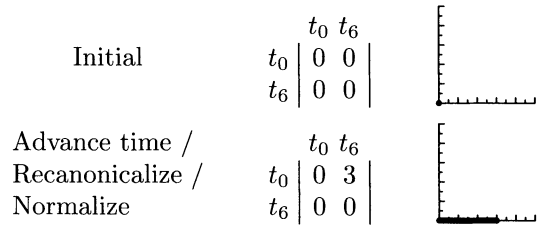


Fig. 7.10 Initial zone.

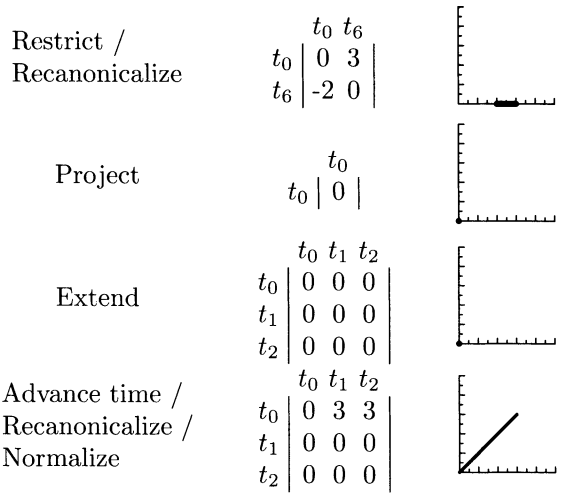


Fig. 7.11 Example of zone creation after the winery is called.

event *Call Winery*, which enables two new rules. So we need to extend the DBM to include the timers t_1 and t_2. We then advance time by setting the 0th row of the DBM to the upper bounds of these timers, three in each case (see Figure 7.11).

In this new zone, there are two possible rule firings. We can fire either the rule r_1 or r_2. Let us first fire r_1. Note that we must remember that we had a choice and come back and consider firing these rules in the other order. Since the lower bound on this rule is 2, we must restrict timer t_1 to be at least 2. This matrix is not maximally tight, so we must recanonicalize. The entry for timer t_2 in the 0th column becomes -2, since from t_2 to t_1 is 0 and t_1 to t_0 is -2. Next, we eliminate the columns corresponding to timer t_1. This rule firing results in the event *Wine Arrives*, so a new timer, t_4, is introduced. We extend the DBM to include t_4. We initialize its lower and upper bounds to 0 (i.e., $m_{40} = m_{04} = 0$). The entry m_{42} is filled in with 3 since timer t_2 could be as large as 3 (i.e., $m_{02} = 3$). The entry m_{24} is filled in with -2 since timer t_2 is at least 2 (i.e., $m_{20} = -2$). Next, we advance

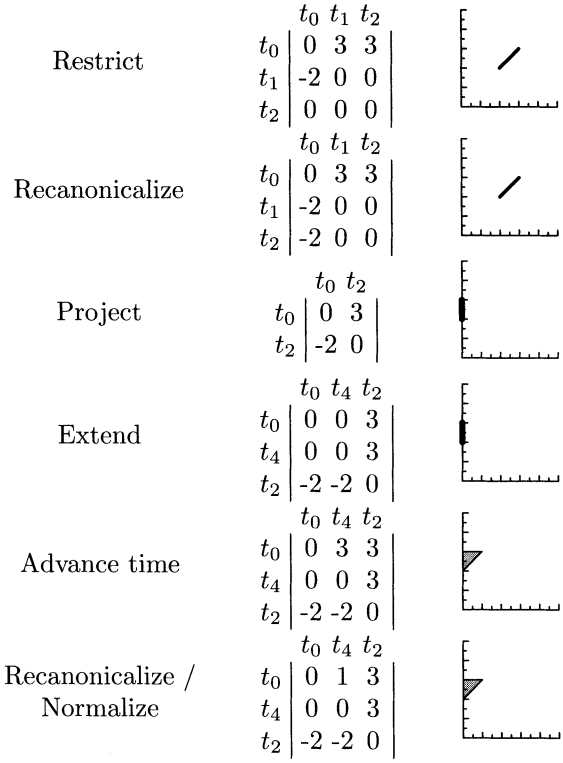


Fig. 7.12 Example of zone creation after the wine arrives.

time by setting the 0th row to the maximum value of all the timers. After recanonicalization, though, the entry m_{04} is reduced to 1, since timer t_4 cannot advance beyond 1 without forcing timer t_2 to expire. In other words, the constraint between t_0 and t_2 is 3 and t_2 and t_4 is -2, which sum to 1. There are no enabled rules with infinite maximums, so normalization has no effect (see Figure 7.12).

In this zone, there are two enabled rules, but only one of the rules is satisfied and can fire. The timer t_4 has a maximum value of 1 (see entry m_{04} in Figure 7.13), so it is not satisfied. Therefore, the only rule that can fire is r_2. Timer t_2 is already at its lower bound, so the restrict and first recanonicalization have no effect. Next, we eliminate the row and column associated with timer t_2. The firing of this rule results in the event *Call Patron*, which enables a new rule, so we must extend the DBM with the new timer t_3. The new entries are all 0 except that m_{34} becomes 1 since timer t_4 may be as high as 1 at this point (see m_{04}). We again advance time by introducing the upper bounds into the 0th row. In this case, one of these entries is infinity. However, after recanonicalization, this infinity is reduced to 3 since the timer t_3 cannot exceed 3 without forcing timer t_4 to expire. There is now a rule with an

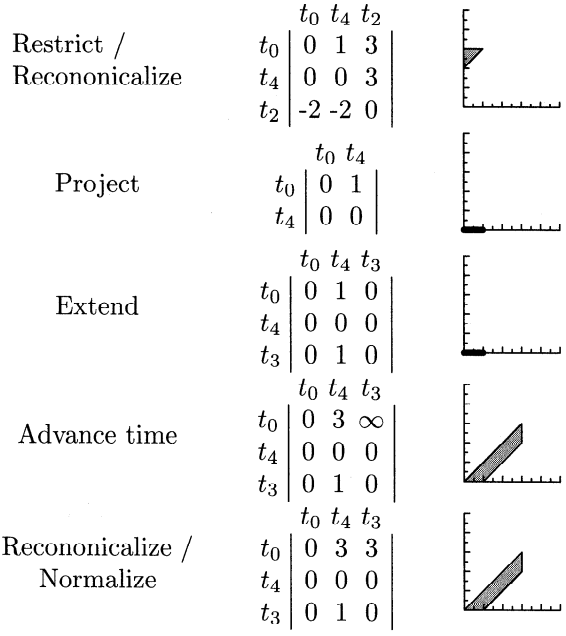


Fig. 7.13 Example of zone creation after wine arrives and patron has been called.

infinite maximum, r_3, but it has not yet reached its lower bound, and so normalization has no effect. These steps are shown in Figure 7.13.

In this zone, again only one of the two enabled rules can fire. The rule r_3 cannot fire since the upper bound of its timer is only 3. Therefore, the only rule that can fire is r_4. To fire this rule, we first restrict timer t_4 to be at least 2, the lower bound of this rule. After recanonicalization, the m_{30} entry becomes -1, indicating that timer t_3 must be at least 1 at this point. Next, we project out the rows and columns associated with timer t_4. Note that the firing of this rule does not result in the occurrence of any event since the rule r_5 has not fired yet. Therefore, no new timers need to be introduced. We advance time by setting the entry m_{03} to ∞, and recanonicalize. The timer t_3 has now exceeded its lower bound, and it has an infinite upper bound. Therefore, normalization changes this timer back to its lower bound. The result is shown in Figure 7.14.

In this zone, the only possible rule that can fire is r_3. We restrict the timer t_5 to its lower bound, 5, and recanonicalize. We then eliminate the row and column associated with timer t_5. The firing of this rule results in the event *Patron Arrives*, which enables a new rule, so we need to extend the DBM to include the new timer t_5. We then advance time on this timer, recanonicalize, and normalize (see Figure 7.15).

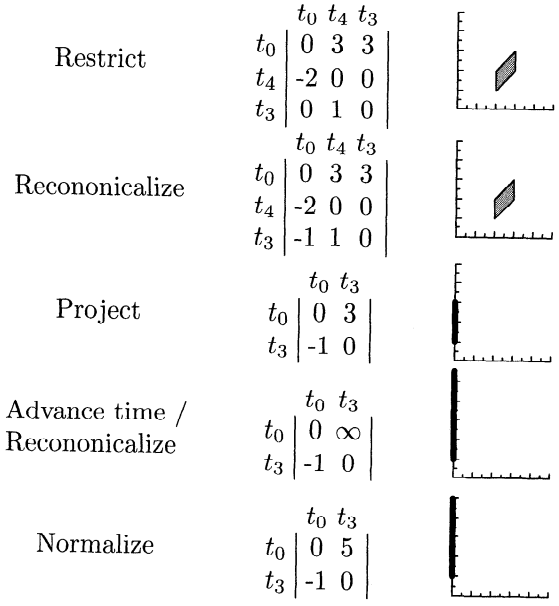


Fig. 7.14 Example of zone creation after a rule expires.

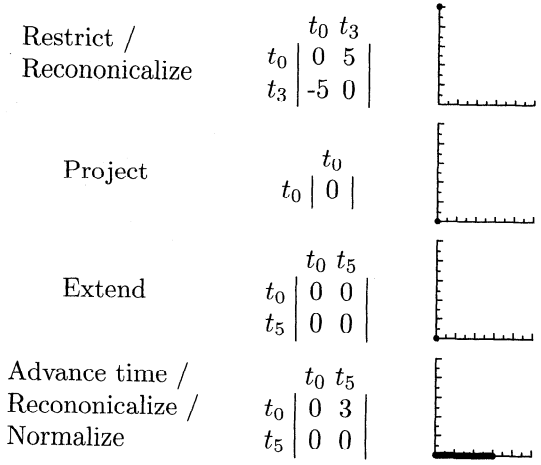


Fig. 7.15 Example of zone creation after the patron arrives.

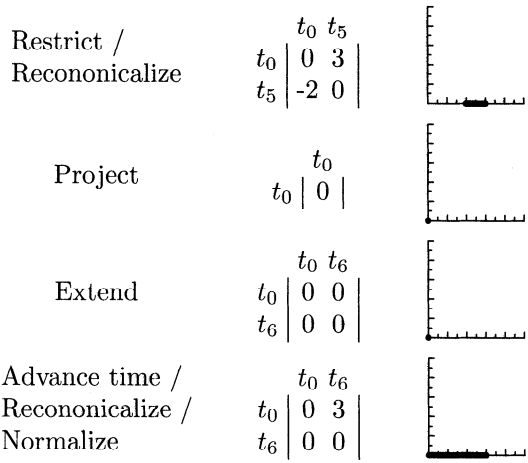
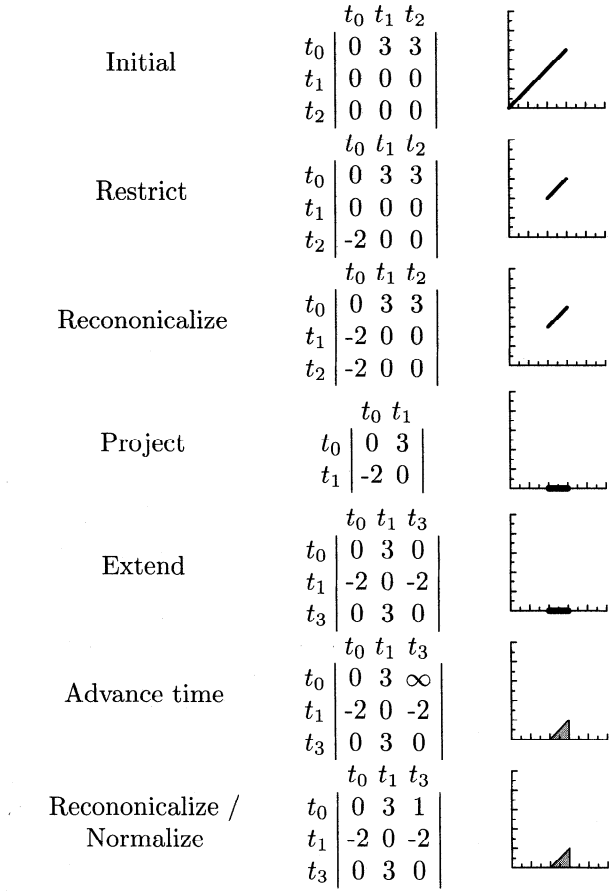


Fig. 7.16 Example of zone creation after the wine is purchased.

In this zone, the rule r_5 fires. We restrict the zone, recanonicalize, and project out timer t_5 (see Figure 7.16). The firing of this rule results in the event *Wine Is Purchased*, since the wine has already arrived and the corresponding rule has fired. Therefore, we extend the DBM to include the newly started timer t_6, we advance time, recanonicalize, and normalize (see Figure 7.16). Note that this timed state matches our initial time state, so we are done exploring down this path.

Remember that earlier we had two satisfied rules, and we chose to let the wine arrive before we called the patron. We must now backtrack to this point and consider the alternative order. This time we call the patron before the wine arrives. In other words, we fire the rule r_2 first (see Figure 7.17). This time, we restrict timer t_2 to its lower bound, 2. We then recanonicalize and project out timer t_2. The event *Call Patron* occurs as a result of this rule firing, which enables a new rule. We must extend the DBM to include the new timer, t_3. The entry m_{13} is set to -2 since timer t_1 is at least 2 at this point, and entry m_{31} is set to 3 since timer t_2 is no more than 3. We then advance time, recanonicalize, and normalize (see Figure 7.17).

In this zone, we can only fire the rule r_1. Its timer is already at its lower bound, so the restrict and first recanonicalization have no effect. We then project out the row and column associated with timer t_1. The event *Wine Arrives* fires, enabling a new rule and introducing the timer t_4. We extend the DBM to include this new timer. All new entries are set to 0 except that m_{43} is 1 since timer t_3 may be as high as 1. We then advance time and recanonicalize (see Figure 7.18). Normalization has no effect since t_3 has not yet reached its lower bound.



Initial		t_0	t_1	t_2
	t_0	0	3	3
	t_1	0	0	0
	t_2	0	0	0

Restrict		t_0	t_1	t_2
	t_0	0	3	3
	t_1	0	0	0
	t_2	-2	0	0

Recononicalize		t_0	t_1	t_2
	t_0	0	3	3
	t_1	-2	0	0
	t_2	-2	0	0

Project		t_0	t_1
	t_0	0	3
	t_1	-2	0

Extend		t_0	t_1	t_3
	t_0	0	3	0
	t_1	-2	0	-2
	t_3	0	3	0

Advance time		t_0	t_1	t_3
	t_0	0	3	∞
	t_1	-2	0	-2
	t_3	0	3	0

Recononicalize / Normalize		t_0	t_1	t_3
	t_0	0	3	1
	t_1	-2	0	-2
	t_3	0	3	0

Fig. 7.17 Example of zone creation after the patron is called.

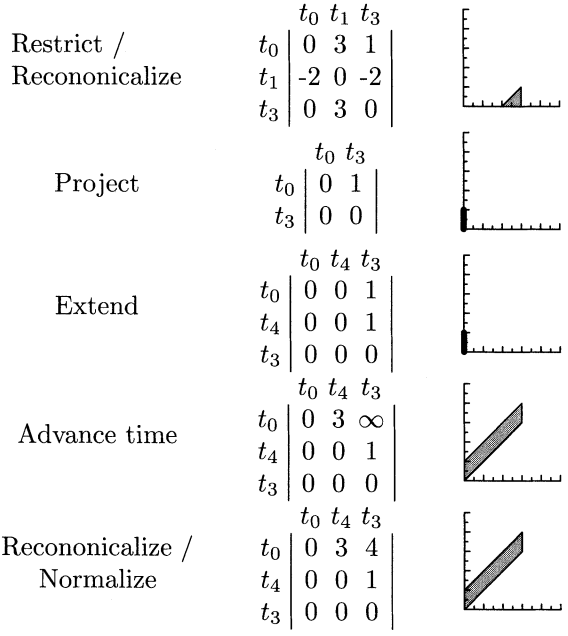


Fig. 7.18 Example of zone creation after the patron has been called and wine arrives.

In this zone, we can only fire the rule r_4. We restrict the timer t_4 to be at least 2, recanonicalize, and project out the timer t_4 (see Figure 7.19). We then advance time and again recanonicalize. Since t_3 has now exceeded its lower bound, normalization brings it back down to its lower bound. The resulting DBM is shown at the bottom of Figure 7.19. Compare this with the one shown at the bottom of Figure 7.14. This new DBM represents a zone that is a subset of the one we found before. This can be seen either in the picture or by the fact that all the entries in the DBM are less than or equal to those found before. Therefore, we do not need to continue state space exploration beyond this point because any possible future would have been found by our exploration beginning with the zone found in Figure 7.14.

Since there are no more unexplored paths, the entire timed state has been found, and it is shown in Figure 7.20. It takes only eight zones to represent the entire timed state space, whereas it took 26 regions or 13 discrete-time states to represent just a part of it. Another important consideration is that if we change the timers such that bounds of $[2, 3]$ are set to $[19, 31]$ and the bounds of $[5, \infty]$ are set to $[53, \infty]$, the number of timed states does not change when represented as zones.

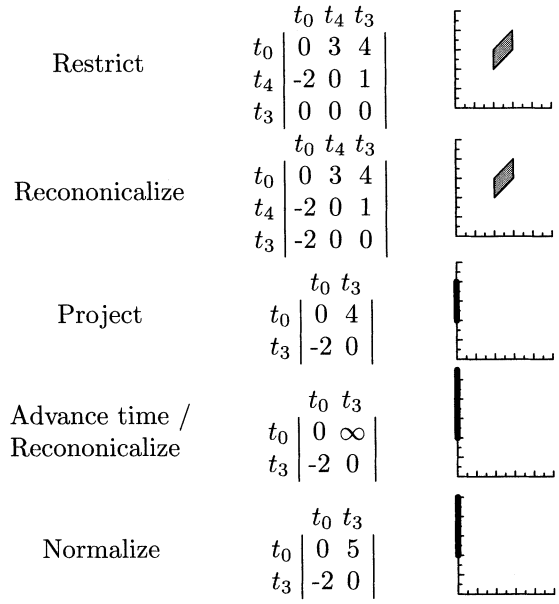


Fig. 7.19 Example of zone creation after a rule expires.

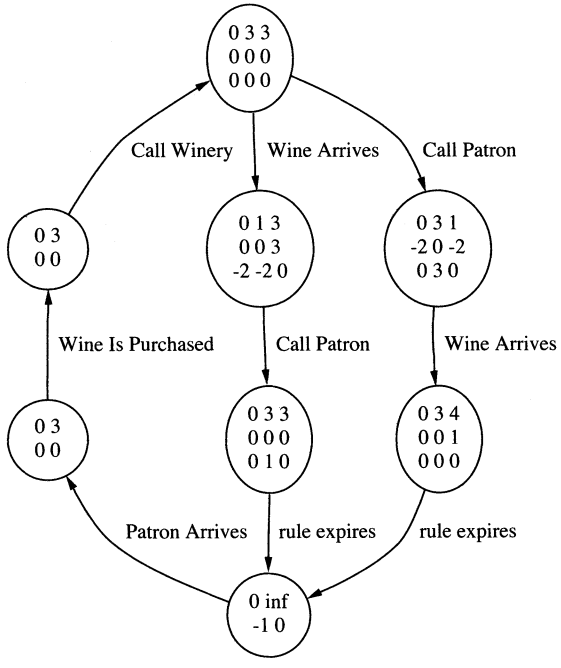


Fig. 7.20 Timed state space using zones.

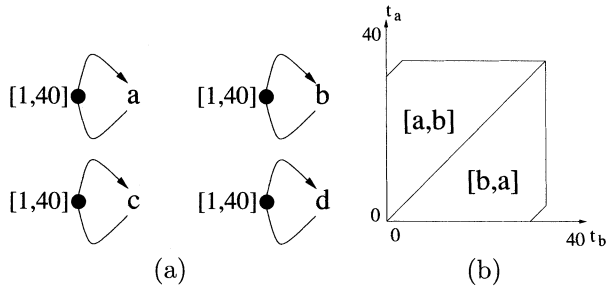


Fig. 7.21 (a) Adverse example. (b) Zones for two firing sequences, $[a, b]$ and $[b, a]$.

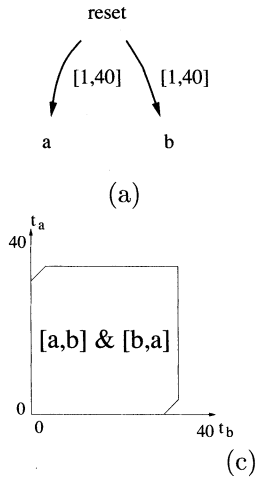
7.5 POSET TIMING

The zone approach works well for a lot of examples, but when there is a high degree of concurrency, there can be more zones than discrete-time states. Consider the simple example shown in Figure 7.21(a). This example obviously has only one untimed state. There are an astronomical 2,825,761 discrete-time states. Even worse, there are 219,977,777 zones!

The reason for the explosion in the number of zones can be illustrated by considering zones found for two possible sequences, $[a, b]$ and $[b, a]$, shown in Figure 7.21(b). The upper zone is found for the sequence $[a, b]$, while the lower zone is found for the sequence $[b, a]$. Even though these two sequences result in the same untimed state, they result in different zones. The zone reflects the order in which the two concurrent events occurred in the sequence. In fact, as the length of the sequence, n, increases, the number of zones grows like $n!$. When linear sequences of events are used to find the timed state space, it is not possible to distinguish concurrency from causality.

In order to separate concurrency from causality, *POSET timing* finds the timed state space by considering *partially ordered sets* (POSETs) of events rather than linear sequences. A graphical representation of a POSET is shown in Figure 7.22(a). This POSET represents both the sequence $[a, b]$ and the sequence $[b, a]$. For each POSET we derive a *POSET matrix* which includes the time separation between each pair of events in the POSET. The POSET matrix shown in Figure 7.22(b) indicates that a and b can occur between 1 and 40 time units after the reset event, r, and that either a or b could occur as much as 39 times units after the other.

After a rule is fired during timed state space exploration using zones and POSETs, the algorithm shown in Figure 7.23 is used to update the POSET matrix and find the corresponding zone. This algorithm takes the old POSET matrix and zone, the rule that fired, a flag that indicates if an event has fired, and the set of enabled rules. If the rule firing resulted in an event firing, we must update the POSET matrix and derive a new zone. First, the newly fired event, f_j, is added to the POSET matrix. Timing relationships are added



	r	a	b
r	0	40	40
a	-1	0	39
b	-1	39	0

(b)

	t_0	t_a	t_b
t_0	0	40	40
t_a	0	0	39
t_b	0	39	0

Fig. 7.22 (a) POSET graph. (b) POSET matrix for sequences $[a, b]$ and $[b, a]$. (c) Zone found for either the sequence $[a, b]$ or $[b, a]$ using the POSET matrix.

between this event and all other events in the POSET matrix. If an event, e_i, in the POSET matrix is *causal* to event f_j, the separation is set to the bounds (i.e., $p_{ji} = -l$ and $p_{ij} = u$). We say that an event e_i is causal to another event f_j if this event is the enabling event for the last rule that fires before f_j fires. If event e_i must directly precede f_j but it is not causal to e_j, the minimum separation is set to the lower bound and the maximum left unbounded (i.e., $p_{ji} = -l$ and $p_{ij} = \infty$). If e_i does not precede f_j directly, no timing relationship is established between these events. The algorithm then recanonicalizes the new POSET matrix and projects out any events that are no longer needed. Events can be projected once there no longer exists any enabled rules that have that event as an enabling event. Next, the algorithm uses the POSET matrix, P, to derive the zone. The algorithm begins by setting the minimums to 0 (i.e., $m_{i0} = 0$) and setting the maximums to the upper bound (i.e., $m_{0j} = u_j$). It then copies the relevant time separations from the POSET matrix to the zone. Consider two timers t_i and t_j, which correspond to two rules with enabling events e_i and e_j, respectively. The m_{ij} entry is found by copying the p_{ij} entry from the POSET matrix. This new zone is then recanonicalized and normalized. If the rule firing did not result in an event firing, the algorithm simply updates the previous zone by projecting out the rule that fired, advancing time, recanonicalizing, and normalizing.

The zone for the POSET in Figure 7.22(b) is shown in Figure 7.22(c). Note that the zone now includes both zones that had been found previously. In fact, if we use this approach to analyze the example in Figure 7.21, we find exactly one zone for the one untimed state.

```
update_poset(P, M, r_j, event_fired, R_en) {
  if (event_fired) then {
    foreach e_i ∈ P                                  /* Update POSET matrix.*/
      if e_i is causal to f_j then {
        p_ji = −l;
        p_ij = u;
      } else if e_i directly precedes e_j then {
        p_ji = −l;
        p_ij = ∞;
      } else {
        p_ji = ∞;
        p_ij = ∞;
      }
    recanonicalize(P);                          /* Tighten loose bounds.*/
    project(P);                    /* Project events no longer needed.*/
    foreach r_i ∈ R_en {                                /* Create a zone.*/
      m_i0 = 0;                                  /* Set minimums to 0.*/
      m_0i = u_i;                          /* Set maximums to upper bound.*/
      foreach r_j ∈ R_en
        m_ij = p_ij;          /* Copy relevant timing from POSET matrix.*/
  } else {
    project(M, r_j);          /* Project out timer for rule that fired.*/
    foreach r_i ∈ R_en                                  /* Advance time.*/
      m_0i = u_i;
  }
  recanonicalize(M);                            /* Tighten loose bounds.*/
  normalize(M, R_en);                  /* Adjust infinite upper bounds.*/
}
```

Fig. 7.23 Algorithm to update the POSET and zone.

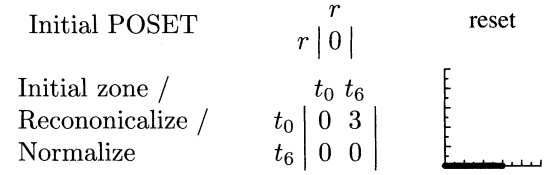


Fig. 7.24 Initial zone using POSETs.

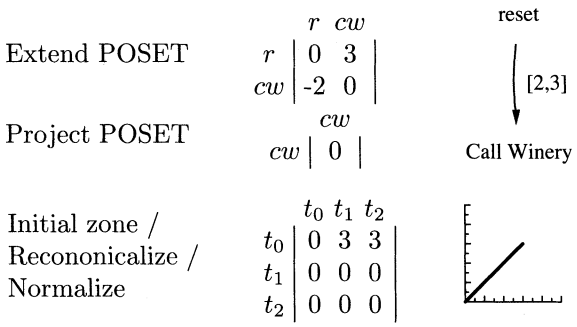


Fig. 7.25 Example of zone creation after the winery is called using POSETs.

Example 7.5.1 Let us again consider the example shown in Figure 7.1. The initial POSET includes only the special *reset* event, so the initial POSET matrix is only one-dimensional. The initial state has only one timer t_6, so our initial DBM is only two by two (i.e., t_0 and t_6). We set the top row entry for t_6 to the maximum value allowed for t_6, which is 3, and the other entries are set to 0. Recanonicalization and normalization result in the same DBM. This sequence of steps is shown in Figure 7.24.

The only possible rule that can fire in this initial zone is r_6. The firing of this rule results in the event *Call Winery*, so we update the POSET to include this event (i.e., *cw*). We no longer need to keep the reset event in the POSET matrix, since all enabled rules have *Call Winery* as their enabling event. Therefore, the POSET matrix is again the trivial one-dimensional matrix. The new zone includes timers t_1 and t_2. We set the 0th row of the DBM to the upper bounds of these timers, three in each case. This sequence of steps is shown in Figure 7.25.

In this new zone, there are two possible rule firings: r_1 or r_2. Let us first fire r_1. We extend the POSET matrix to include *Wine Arrives* (i.e., *wa*), which can happen between 2 and 3 time units after *Call Winery*. We create a zone including the two timers t_4 and t_2. We fill in the upper bounds in the top row, and copy the information from the POSET matrix for the core. We then recanonicalize and normalize. The result is shown in Figure 7.26.

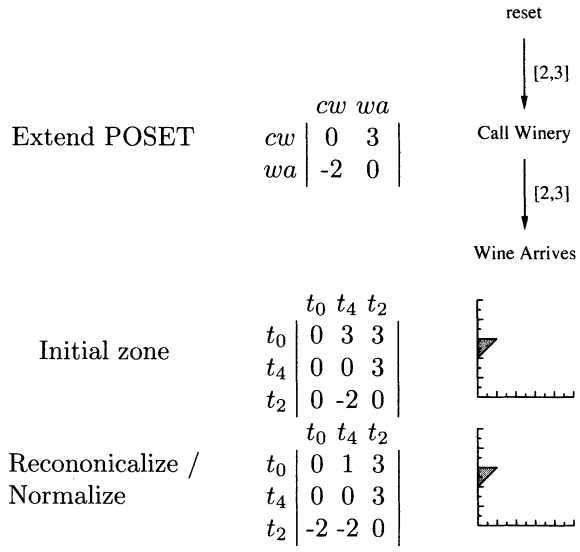


Fig. 7.26 Example of zone creation after the wine arrives using POSETs.

In this zone, the only satisfied rule is r_2. The firing of this rule results in the event *Call Patron.* We add *Call Patron* (i.e., *cp*) to the POSET matrix. After recanonicalization, we determine that either *Call Patron* or *Wine Arrives* can occur up to 1 time unit after the other. We can now remove *Call Winery* from the POSET matrix since it is no longer an enabling event for any active rules. We create a zone including the active timers t_4 and t_3. We copy the upper bounds into the top row, determine the core from the POSET matrix, recanonicalize, and normalize. The result is shown in Figure 7.27. This zone is the union of the two shown in Figures 7.13 and 7.18. In other words, even though we found this by considering that *Wine Arrives* first, it produces a zone that includes the case where the *Call Patron* event happens first.

In this zone, the only rule that can fire is r_4. The firing of this rule does not result in an event firing. Therefore, we simply update the zone. The zone is updated by projecting t_4 from the zone shown at the bottom of Figure 7.27, advancing time, recanonicalizing, and normalizing. The result is shown in Figure 7.28.

In this zone, the only possible rule that can fire is r_3. This results in the event *Patron Arrives*, which must be added to the POSET matrix, and we recanonicalize the POSET matrix. The events *Wine Arrives* and *Call Patron* can be removed from the matrix since they are not needed for any active rules. We then create a zone for the one active timer t_5 as shown in Figure 7.29.

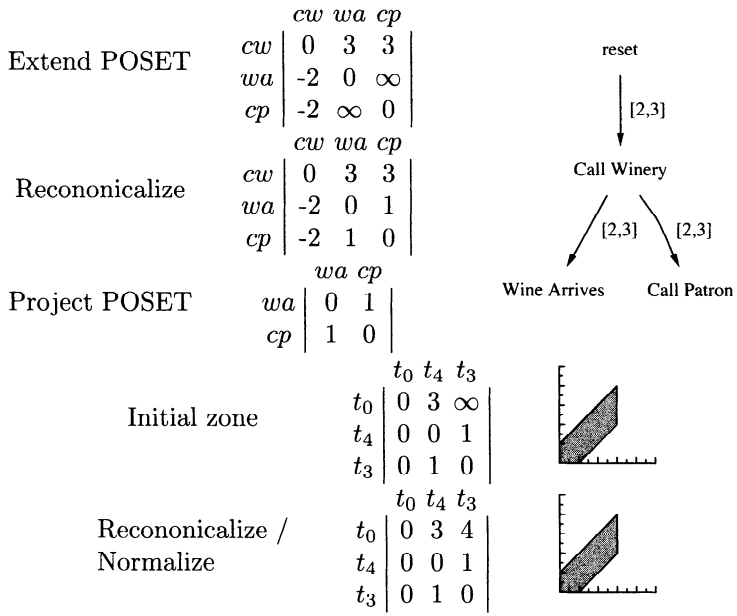


Fig. 7.27 Example of zone creation after wine arrives and patron has been called using POSETs.

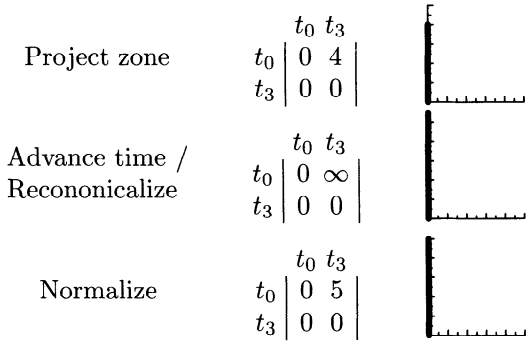


Fig. 7.28 Example of zone creation after a rule expires using POSETs.

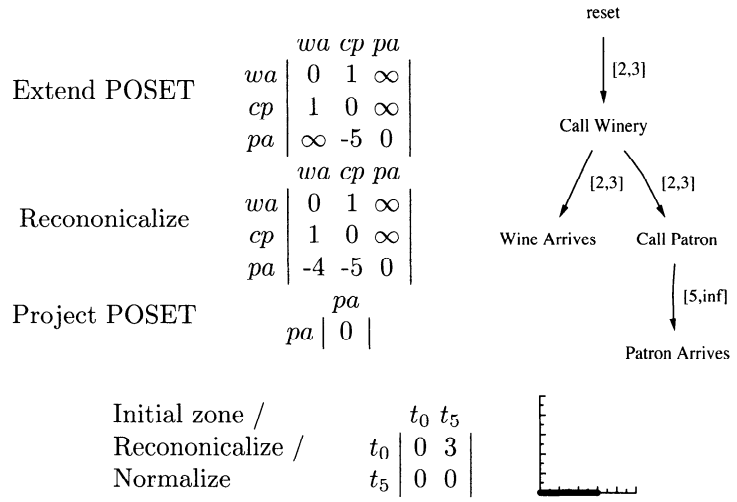


Fig. 7.29 Example of zone creation after the patron arrives using POSETs.

In this zone, the rule r_5 fires. The result is that we add the *Wine Is Purchased* event to the POSET matrix and remove the event *Patron Arrives*. We then derive a new zone for the active timer t_6, as shown in Figure 7.30. The resulting zone is the same as the initial zone, so we are done exploring down this path.

We now backtrack and consider calling the patron before the wine arrives. The POSET for this case is shown in Figure 7.31. Using the POSET matrix, we derive the new zone, recanonicalize, and normalize as shown in Figure 7.31.

In this zone, we fire the rule r_1. This result, shown in Figure 7.32, is the same POSET in Figure 7.27. Therefore, as expected, it results in the same zone. We can now backtrack at this point. Note that we get to backtrack one state sooner than when we used zones alone.

Since there are no more alternative paths that we have not explored, we have completed the timed state space exploration. The entire timed state space is shown in Figure 7.33. Using POSETs, we found seven zones to represent the entire timed state space, compared with the eight we found before. Although this is a very modest improvement, for highly concurrent examples the amount of improvement can be orders of magnitude.

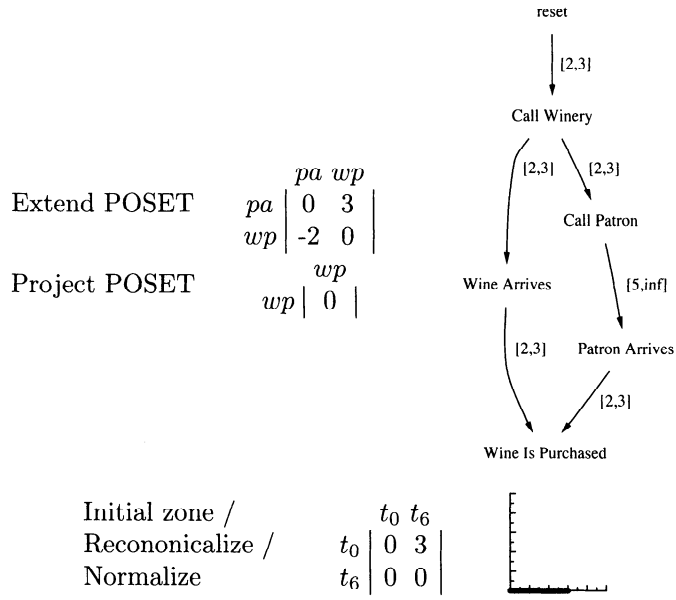


Fig. 7.30 Example of zone creation after the wine is purchased using POSETs.

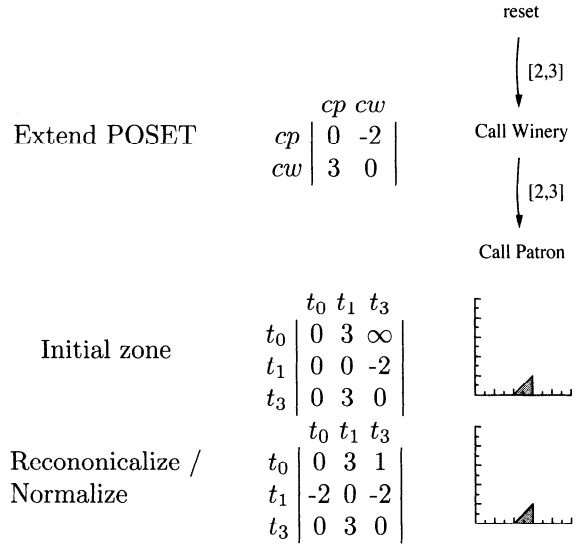


Fig. 7.31 Example of zone creation after the patron is called using POSETs.

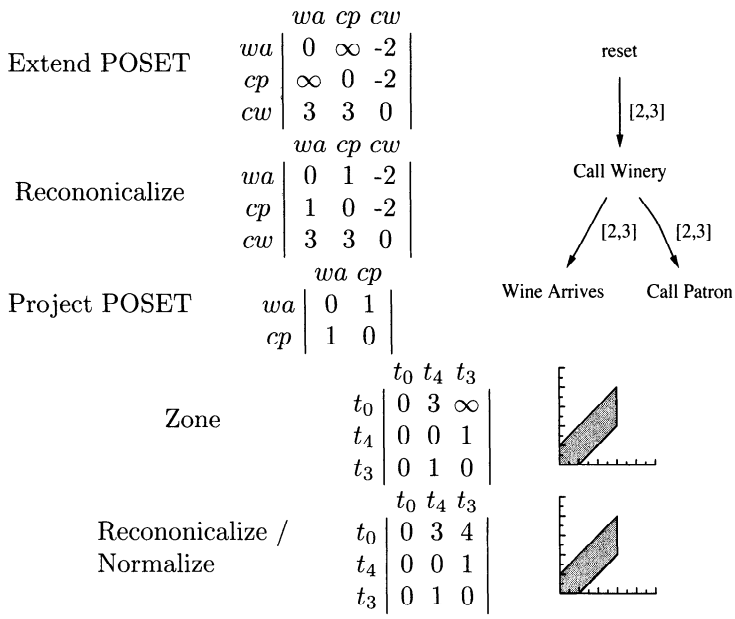


Fig. 7.32 Example of zone creation after the patron has been called and wine arrives using POSETs.

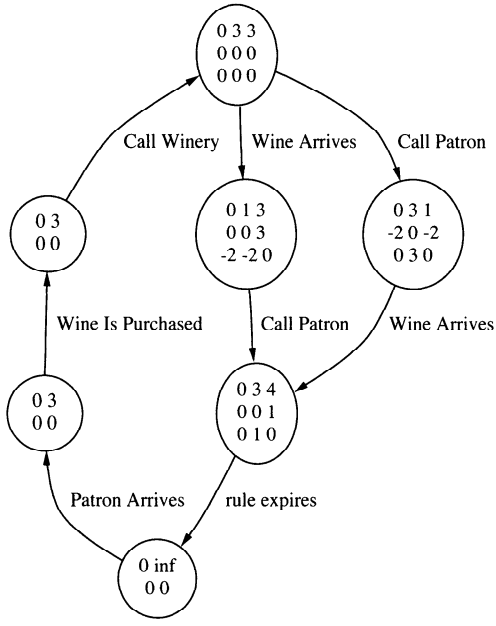


Fig. 7.33 Timed state space using POSETs.

Table 7.1 Timing assumptions for passive/active wine shop.

Assumption	Delay
Winery delays	2 to 3 minutes
Patron responds	5 to ∞ minutes
Patron resets	2 to 3 minutes
Inverter delay	0 to 6 seconds
AND gate delay	6 to 12 seconds
OR gate delay	6 to 12 seconds
C-element delay	12 to 18 seconds

7.6 TIMED CIRCUITS

We conclude this chapter by showing the effect that timing information can have on the timed circuit that is synthesized.

Example 7.6.1 We return to the passive/active wine shop example discussed in Chapter 6. It was found at the end of Chapter 6 that if the bubbles at the inputs to the AND gates in the circuit shown in Figure 6.23 are changed to explicit inverters, the circuit is no longer hazard-free under the speed-independent delay model. However, if we make the timing assumptions shown in Table 7.1 (granted a pretty slow logic family), we can determine using timed state space exploration that this circuit is hazard-free.

Not only can we verify that our gate-level circuit is hazard-free, we can use the timing assumptions to further improve the quality of the design. We begin using a TEL structure shown in Figure 7.34. We assume the delays given in Table 7.1 for the winery and patron and that the total circuit delays for any output are never more than 1 minute.

If we ignore the timing information provided, we would find the state graph shown in Figure 7.35, which has 18 states. Using the timing information, we derive a state graph that has only 12 states, as shown in Figure 7.36. Using the speed-independent state graph, the resulting circuit implementation is shown in Figure 7.37. The timed circuit is shown in Figure 7.38. The timed circuit implementation for *CSC0* is the same and for *req_patron* is reduced by one literal. The circuit for *ack_wine* is substantially reduced to only a three-input AND gate. Furthermore, we do not need to assume that we have a library of AND gates with inverted inputs, as the design using explicit inverters is also hazard-free.

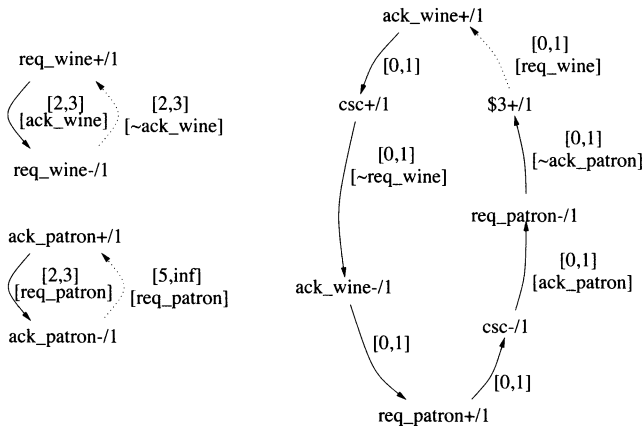


Fig. 7.34 TEL structure for the wine shop example.

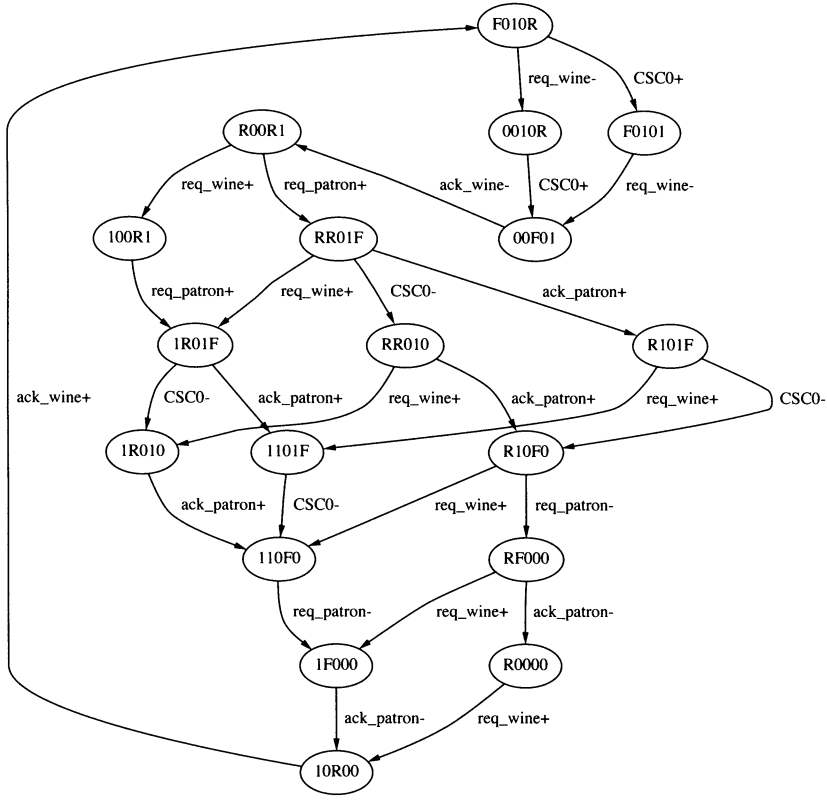


Fig. 7.35 Speed-independent state graph for the wine shop example (state vector is $\langle req\_wine, ack\_patron, ack\_wine, req\_patron, CSC0 \rangle$).

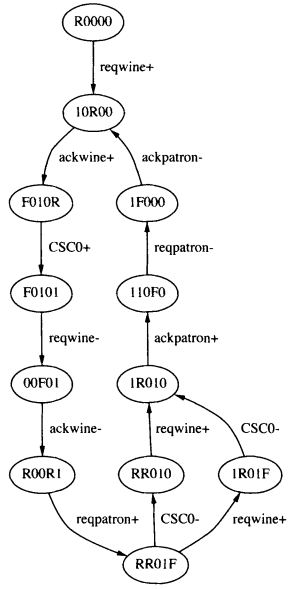


Fig. 7.36 Reduced state graph for the wine shop example (state vector is $\langle req\_wine, ack\_patron, ack\_wine, req\_patron, CSC0 \rangle$).

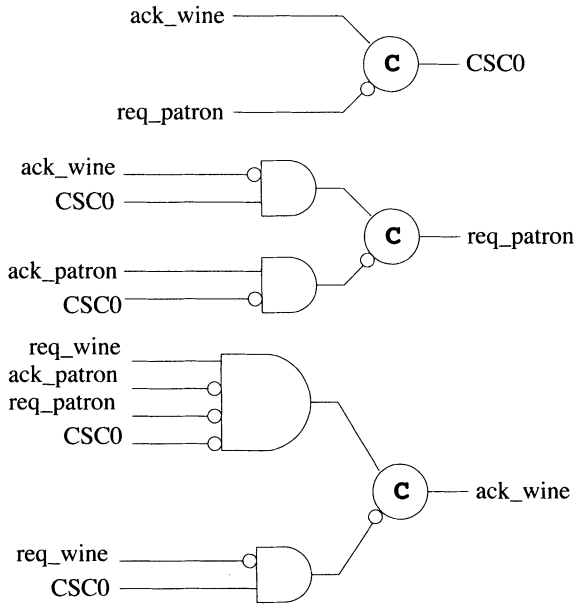


Fig. 7.37 Speed-independent circuit for the wine shop example.

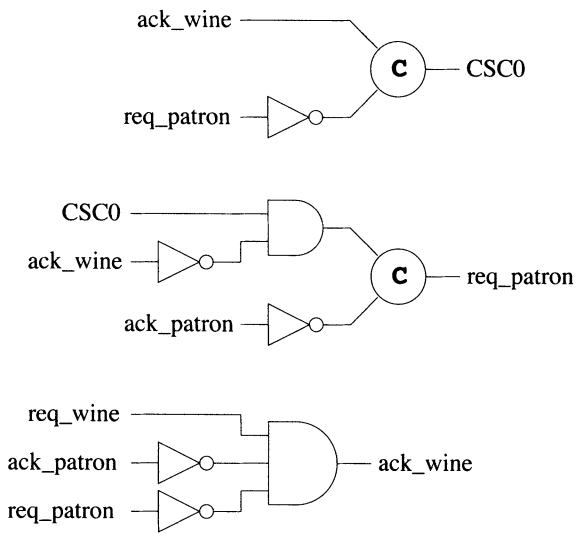


Fig. 7.38 Timed circuit for the wine shop example.

7.7 SOURCES

The region-based timing analysis method described in Section 7.2 was developed by Alur et al. [7, 8, 9]. The discrete-time analysis technique described in Section 7.3 was first applied to timed circuits by Burch [62, 63]. Recently, Bozga et al. proposed a discrete-time analysis approach which utilized binary decision diagrams [45]. Dill devised the zone method of representing timed states described in Section 7.4 [114]. The KRONOS tool [417] has been applied to timed circuit analysis, and it includes both the discrete-time method from [45] and a zone-based method. Rokicki and Myers developed the POSET timing analysis method described in Section 7.5 [325, 326]. Belluomini et al. generalized the POSET method to apply to TEL structures and applied it to the verification of numerous timed circuits and systems from both synchronous and asynchronous designs [31, 32, 33, 34]. An implicit method using multiterminal BDDs was proposed by Thacker et al. [377]. Myers et al. first applied timed state space exploration to produce optimized timed asynchronous circuits as described in Section 7.6 [283, 285, 286, 287].

A different approach to timing analysis and optimization is based on the idea of *time separation of events* (TSE). A TSE analysis algorithm finds the minimum and maximum separation between any two specified events. This information can later be used during state space exploration to determine whether or not two concurrently enabled events can occur in either order or must be ordered. McMillan and Dill developed a TSE algorithm for acyclic graphs [265]. Myers and Meng utilized a similar algorithm to determine an

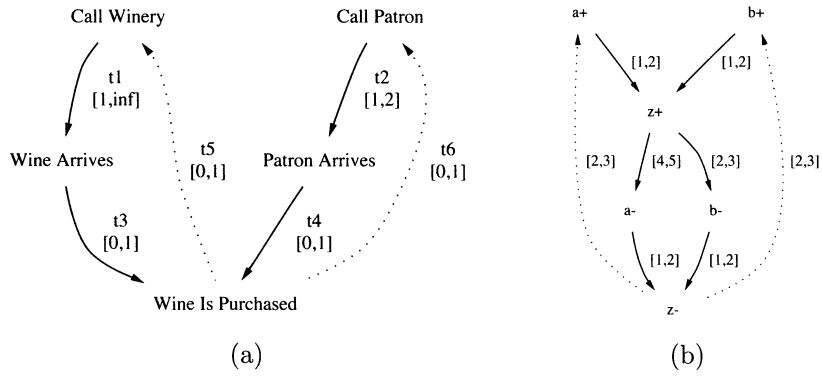


Fig. 7.39 TEL structures for homework problems.

estimate of the TSE in cyclic, choice-free graphs in polynomial time. They also applied this algorithm to the synthesis of timed circuits [285]. Jung and Myers also used this algorithm in a direct synthesis method from STGs to timed circuits [186]. Hulgaard et al. developed an exact TSE algorithm for cyclic graphs, including some types of choice behavior [173, 174, 175].

Problems

7.1 Modeling Timing

Give two timed firing sequences for the TEL structure shown in Figure 7.39(a) that end with the *Wine Is Purchased* event.

7.2 Modeling Timing

Give two timed firing sequences for the TEL structure shown in Figure 7.39(b) that end with the $z+$ event.

7.3 Regions

Using regions to represent timing, find the timed state space for one timed firing sequence for the TEL structure shown in Figure 7.39(a) that ends with the *Wine Is Purchased* event.

7.4 Regions

Using regions to represent timing, find the timed state space for one timed firing sequence for the TEL structure shown in Figure 7.39(b) that ends with the $z+$ event.

7.5 Discrete Time

Using discrete-time states, find the timed state space for one timed firing sequence for the TEL structure shown in Figure 7.39(a) that ends with the *Wine Is Purchased* event.

7.6 Discrete Time

Using discrete-time states, find the timed state space for one timed firing sequence for the TEL structure shown in Figure 7.39(b) that ends with the $z+$ event.

7.7 Zones

Using zones to represent timing, find the entire timed state space for the TEL structure shown in Figure 7.39(a).

7.8 Zones

Using zones to represent timing, find the entire timed state space for the TEL structure shown in Figure 7.39(b).

7.9 POSET Timing

Using POSETs and zones, find the entire timed state space for the TEL structure shown in Figure 7.39(a).

7.10 POSET Timing

Using POSETs and zones, find the entire timed state space for the TEL structure shown in Figure 7.39(b).

7.11 Timed Circuits

Consider the TEL structure shown in Figure 7.39(b).

7.11.1. Find a SG ignoring timing and use it to find a Muller circuit for output signal z.

7.11.2. Find a SG considering timing and use it to find a timed circuit for output signal z.