

CHAPTER 2

CIRCUIT SPECIFICATIONS AND SEMANTICS

In this chapter, we expand on the timed asynchronous technology mapping design flow presented in Chapter 1. This expanded design flow is shown in Figure 2.1. We then develop the necessary semantics to support the remainder of this thesis and illustrate with two examples.

Our work begins with the logic equations generated by the synthesis portion of the design flow and ends with a (hopefully) hazard-free netlist of the circuit using only cells available from a standard library.

2.1 Logic Expressions

The logic expressions provided by the synthesis engine are assumed to be of a sum-of-products (SOP) form. Figure 2.2(a) shows the SOP structure that results from synthesis. Although this example shows a limited number of terms, there are an arbitrary number of AND terms and each AND term can have an arbitrary number of literals in the actual implementation. In this form, each output is naturally partitioned into a cone of logic. As a result, there is no partitioning step necessary in this design flow.

In Figure 2.2(a), all AND terms are OR'd together to create the inputs to a C-element. The C-element is a state holding device [41]. When both of its inputs are '0', its output goes to '0'. Similarly, when both of its inputs are '1' its output goes to '1'. Otherwise, it retains its old value. In boolean form, the logic equation can be written as $c = ab + ac + bc$ where a and b are inputs to the C-element and c is the output. Using the C-element is preferable in asynchronous circuits over an SR latch for state holding circuits because it does not exhibit metastability in any of

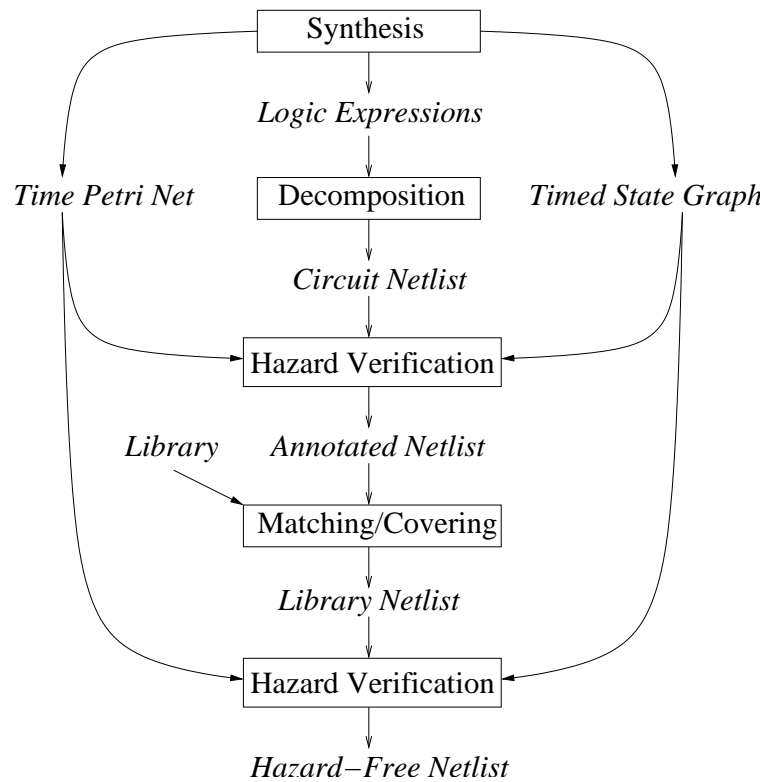


Figure 2.1. Expanded timed asynchronous technology mapping design flow.

its states. The upper sum-of-products represents the logic that sets the output of the C-element high and the lower sum-of-products represents the logic that resets the output. The set and reset cones of logic are naturally de-coupled and can be decomposed separately unless gate sharing is enabled. If there are only set terms present or there are only reset terms present then the output is combinational and the C-element can be removed.

A generalized C-element (gC) is a state holding component that, in addition to containing the C-element, also contains the set and reset logic. Figure 2.2(b) shows a transistor level gC implementation of the logic shown in Figure 2.2(a). This implementation, with weak feedback, is considerably more efficient than a fully static implementation. An attractive feature of gC's is that they reduce the potential for hazards. Static hazards cannot manifest on the output of a gC gate

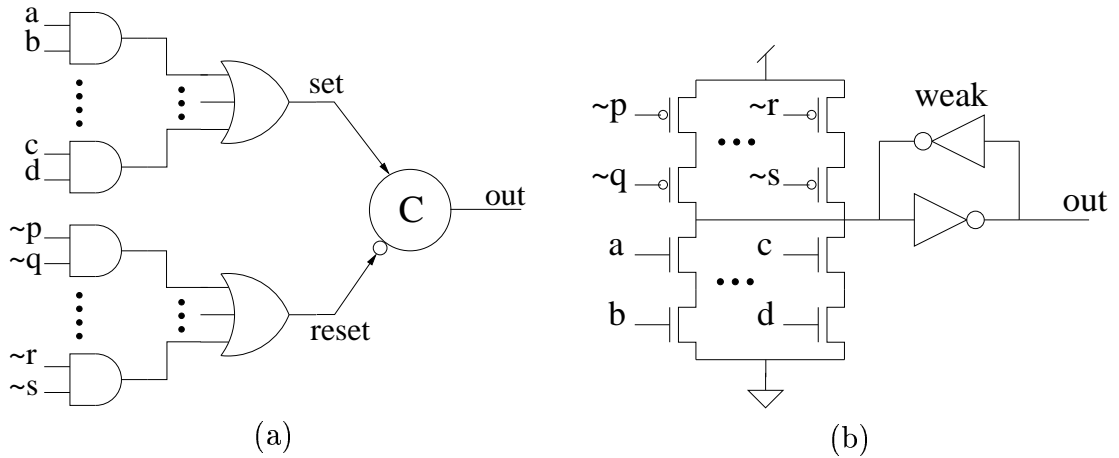


Figure 2.2. (a) POS form of the logic expressions. (b) A CMOS implementation with weak feedback.

by its very structure. However, the synthesis engine must ensure that there is never both a set and reset term active concurrently or a direct short appears in the CMOS stack.

In actuality, the synthesized equations do guarantee that a short circuit will not occur directly in the transistor stack because this feature corresponds directly to avoiding dynamic hazards caused by decomposing an N- or P-stack[41]. However, this is true only if the equations are implemented in a gC element. Once we change the structure of the netlist through decomposition and cover some portions of the network with library cells other than gC elements, we introduce the possibility of small windows of time where a short circuit can occur. When a gC element is chosen from the library as a best match, a dynamic timing analysis must be done to ensure that a short circuit condition is not present.

2.2 Time Petri Nets

As an intermediate form of circuit representation, the synthesis engine creates a *Time Petri Net* (TPN) to model the possible input behaviors and the required output behaviors for timed circuits [37]. Let W be a finite set of wires in a timed

circuit. The timed behavior of a circuit is modeled as sequences of rising and falling transitions on W . For any $w \in W$, $w+$ is a rising transition and $w-$ is a falling transition on the wire w . In the following definitions, let \mathbb{Q}^+ and \mathbb{R}^+ denote the sets of non-negative rational and non-negative real numbers, respectively. A W -labeled one-safe TPN is a directed bipartite digraph described by the tuple $TPN = \langle W, T, P, F, M_0, s_0, l, u, L \rangle$ where:

- $W = I \cup O$ is the set of wires where I is the set of input wires and O is the set of output wires;
- T is the set of transitions;
- P is the set of places;
- $F \subseteq (T \times P) \cup (P \times T)$ is the flow relation;
- $M_0 \subseteq P$ is the initial marking;
- $s_0 \subseteq W$ is the set of wires that are initially high;
- $l : T \rightarrow \mathbb{Q}^+$ is the lower timing bound function;
- $u : T \rightarrow \mathbb{Q}^+ \cup \{\infty\}$ is the upper timing bound function;
- $L : T \rightarrow W$ is the labeling function.

The state of a TPN is a pair $\langle M, D \rangle$ where M is the current marking (i.e., the subset of places that hold tokens) and $D : T \rightarrow \mathbb{R}^+$ is a clock assignment function assigning nonnegative real valued ages to transitions. With every transition $t \in T$, its associated *preset* is $\bullet t = \{p \in P \mid (p, t) \in F\}$. The *postset* of a transition is defined as $t \bullet = \{p \in P \mid (t, p) \in F\}$. Note that the preset and postset for places are defined in a similar manner. A transition, t , is *enabled* in a state if the members of its preset form a subset of the places in the marking of the state (i.e., $\bullet t \subseteq M$). A transition, t , is *fireable* in a state if it has been enabled longer than its lower timing bound (i.e., $D(t) \geq l(t)$). A transition, t , *must fire* before it has been enabled longer than its upper timing bound (i.e., $D(t)$ must never exceed $u(t)$).

A TPN for the example presented in Figure 1.5 is shown in Figure 2.3(a). In the initial state, transitions $a+$ and $b+/1$ are enabled, and exactly one of these transitions fires within 2 to 5 time units. If $a+$ fires, then the $b+/2$ transition becomes enabled and fires within 2 to 5 more time units enabling $d+$. The sequence thus continues throughout the TPN.

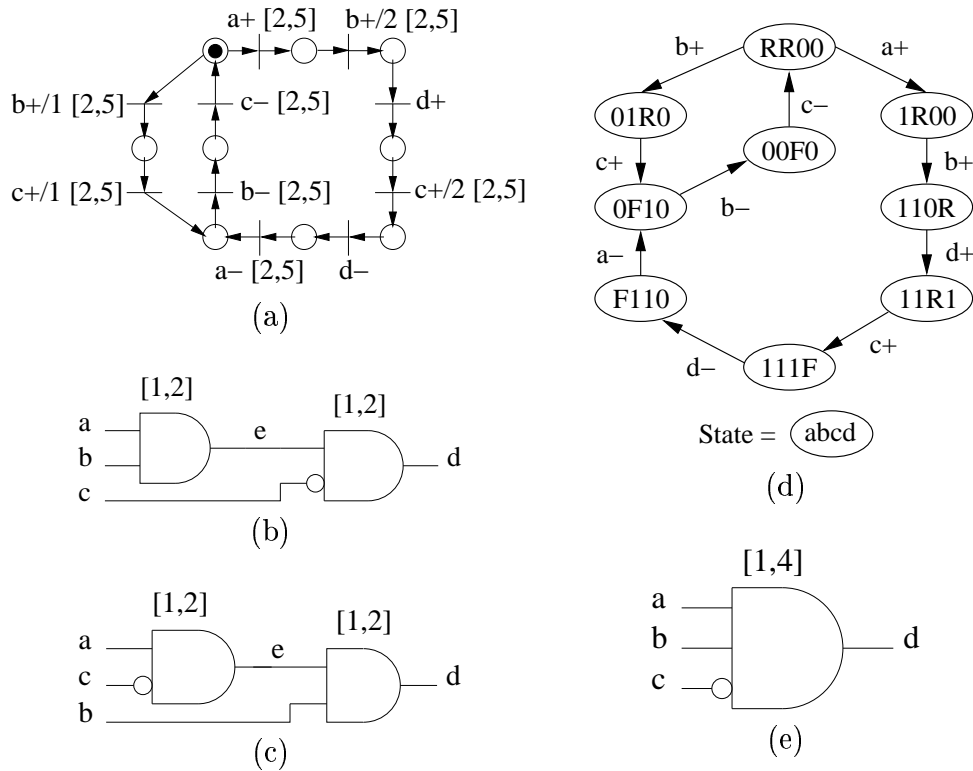


Figure 2.3. (a) An example TPN. (b) A circuit netlist that is hazardous under the speed-independent model. (c) A circuit netlist that is hazard-free under the speed-independent model. (d) A state graph. (e) A complex gate equivalent circuit.

2.3 Netlist

The goal of the verification portion of our work is to verify the correctness of a circuit implementation against a given TPN specification. The circuit to be verified is described using a netlist modeled by a directed graph $NET = \langle V, E \rangle$ where

- $V = I \cup O \cup N$ is the set of vertices in the circuit;
- $E \subseteq (I \cup O \cup N) \times (N \cup O)$ is the edges between vertices.

Each vertex $v \in V$ represents a node in the netlist. This set is composed of both the input wires, I , and output wires, O , from the TPN description as well as new nodes internal to the circuit, N . Each $e \in E$ represents a directed connection in the netlist from one node to another node. The set of *fanins* to a node is denoted by $FI(v)$, and the *fanouts* are denoted by $FO(v)$. Each node which is in $N \cup O$ has an associated gate output function $f_v(v_1, \dots, v_r)$ where $FI(v) = \{v_1, \dots, v_r\}$. This gate output function also has an associated minimum, min_v , and maximum, max_v , gate delay.

The netlist for a possible circuit implementation of the signal d in our example is shown in Figure 2.3(b). The set of vertices, V , is $\{a, b, c, d, e\}$, and the set of edges, E , is $\{(a, e), (b, e), (e, d), (c, d)\}$. The function associated with e is $f_e(a, b) = AND(a, b)$ which has a delay of 1 to 2 time units. An alternative circuit implementation for signal d is shown in Figure 2.3(c).

2.4 State Graphs

In order to check correctness, a verification method typically uses a specification such as a TPN and a representation of the circuit implementation such as a netlist and finds all possible states represented using a state graph. This verification method then checks the state graph (often on the fly as the state graph is being generated) for various correctness properties. We first define the semantics for a state graph and a netlist and then show how the state graph of Figure 2.3(d) can be derived from the TPN of Figure 2.3(a) and the netlist of Figure 2.3(b) or (c).

A SG is a labeled directed graph whose nodes are *states* and edges are *state transitions*. Formally, a SG is modeled by the tuple $SG = \langle S, \delta \rangle$ where

- S is the set of states.
- $\delta \subseteq S \times T \times S$ is the set of state transitions.

Each individual state $s \in S$ is modeled as a tuple $s = \langle \nu, z \rangle$ where

- $\nu \subseteq V$ is the set of wires that are high in the state.
- z is a *zone* representing timing relationships.

Timing information in each state is given using zones which are typically represented using difference bound matrices (DBMs) [15]. These matrices represent time differences between recently fired transitions. Each entry, z_{ij} , in the matrix represents a timing relationship of the form $\tau_{t_i} - \tau_{t_j} \leq z_{ij}$ where τ_{t_i} is the time at which t_i fires. In other words, z_{ij} represents the maximum amount of time in which t_i fires after t_j . An example zone for the point right after $a+$ fires is given below which represents the relationship $2 \leq \tau_{a+} - \tau_{c-} \leq 5$.

$$\begin{array}{c} \tau_{c-} \quad \tau_{a+} \\ \tau_{c-} \left| \begin{array}{cc} 0 & -2 \\ 5 & 0 \end{array} \right. \\ \tau_{a+} \end{array}$$

The states are labeled in the state diagram to show the value of all signal wires in the state. Each edge of the state graph is labeled with a signal transition. The input wire set is $\{a, b, c\}$ and the output wire set is $\{d\}$. There are nine states including $RR00$ and $1R00$, and ten state transitions including $(RR00, a+, 1R00)$.

Using a timed state space exploration algorithm such as the ones in [5, 36], it is possible at this point to derive the state graph of Figure 2.3(d) using the TPN to drive the inputs and check the outputs, and a netlist to drive the outputs. This state graph reflects the *complex gate equivalent* version of the netlists of Figure 2.3(b) and (c) and is shown in Figure 2.3(e) where the only signals present are the primary inputs a, b, c and the primary output d . As the complex gate equivalent circuit of Figure 2.3(e) is decomposed, new internal signals are added and a new state graph must be constructed to include these new signals. The state graph for the netlist of Figure 2.3(c) is shown in Figure 2.4. Note the increased number of states. Every new node added to the circuit potentially doubles the number of states in the state graph. In this particular example, the number of states increased from 9 to 12 reflecting the fact that the synthesis engine pruned some unnecessary states from the state graph. This state graph expansion is a function of circuit structure and it's growth can conservatively be estimated as doubling the number of states for

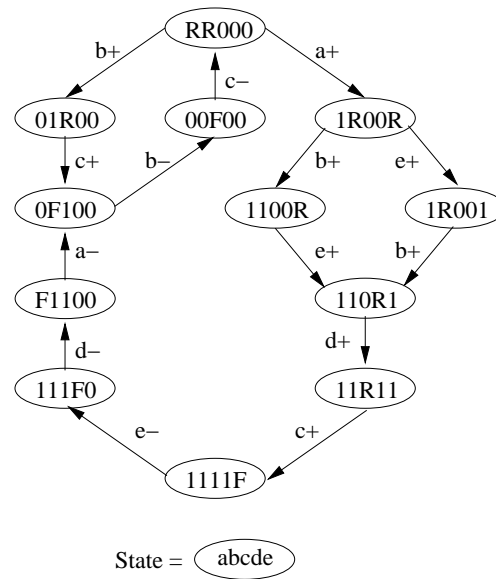


Figure 2.4. State graph for the network of Figure 2.3(c).

each new node added. This growth becomes prohibitively large as a netlist grows in complexity.

The key result of our work is that our method never explicitly derives a new state graph for a decomposed netlist. Instead, we only need to derive a state graph for complex gate equivalent of the network. The details of this procedure are explained in Chapter 3.

2.5 Design Flow Completion

The hazard verification process of Figure 2.1 takes as inputs a *time Petri net* (TPN) defining the circuit and the behavior of the specified environment, a *netlist* representing the decomposed circuit to be verified, and a *state graph* which represents reachable timed states. This process is discussed fully in Chapter 3 where each node in the decomposed netlist is checked for hazard conditions. Each node is then marked as to its hazard properties and this annotated netlist is passed to the matching/covering stage.

The matching/covering stage takes as inputs the annotated netlist from the

hazard verification process along with a technology dependent library and creates a new netlist composed exclusively of cells from the library. The matching stage finds the best match at every node from the available library cells. The matcher takes into consideration a primary and secondary cost factor chosen from hazard-freedom, area, or delay. Since a sub-optimized netlist in regards to area and/or delay still functions correctly provided it is hazard-free, our primary cost objective is hazard-freedom. Without a hazard-free circuit, there is no guarantee of correct operation. Thus the matcher makes a determination at each node of the circuit which library cell best matches the circuit at that node and annotates each node with the selected library cell. The matcher takes into account whether or not a hazard exists on the current node being worked on and whether or not it should be encapsulated or left exposed.

There are two types of hazards that we are dealing with in this process. These two hazard types are more formally defined and explained in Chapter 3. For now, the matcher must make every effort to encapsulate *acknowledgment* hazards and leave uncovered *monotonicity* hazards. An acknowledgment hazard occurs when an internal node becomes excited to change to a new value, but its excitation changes value before it can be shown to have stabilized. A monotonicity hazard occurs when an internal or output node is supposed to remain stable but it becomes momentarily excited or it is supposed to make a transition which it makes non-monotonically.

After the circuit is matched, the covering algorithm determines the best cover using library cells and writes a new, reduced netlist composed only of library cells. For two reasons, it is still possible that our reduced netlist may yet contain some hazardous nodes. First, it may not be possible to eliminate all hazardous nodes in the covering. Second, the new covered netlist alters the timing properties and may not verify in a hazard-free manner against the original timing specifications. Although we have not seen results indicating so, it is certainly feasible that the covered circuit may create new hazards due to new timing properties. As a result, we perform another hazard verification on the library netlist.

If this final verification shows no hazards our technology mapping is complete

and the library netlist is a hazard-free implementation of the original circuit specification. If not, we must iterate on the design flow until we can produce a hazard-free netlist or prove that one is not possible. This iteration process involves a number of optimizations discussed in more detail in Chapter 4.

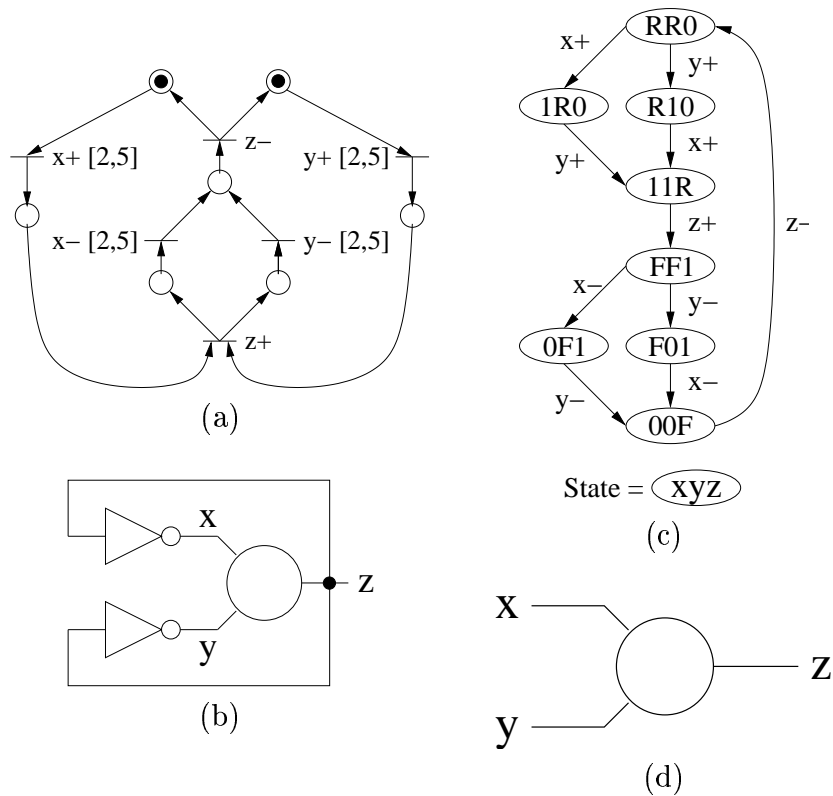


Figure 2.5. (a) A Petri net of a C-element. (b) A netlist of a C-element. (c) A state graph of a C-element. (d) A complex gate equivalent netlist of a C-element.

2.6 Another Example

We conclude this chapter with another example to illustrate the process by which circuit representations are created and how they inter-relate. We begin with a time Petri net and a synthesized netlist, then create a state graph, and finally represent the netlist as a complex gate equivalent circuit.

The example we have chosen is that is that of the basic C-element. An example Petri net for the C-element is shown in Figure 2.5(a). Initially x and y are enabled to change to '1', that is $x+$ and $y+$ are enabled to fire. After both x and y change to '1', z is enabled to change to '1'. Note that which order x and y fire in does not matter. Z cannot fire until both $x+$ and $y+$ have occurred. After z changes to '1', x and y are enabled to change from '1' to '0'. After they have both changed to '0', z can change from a '1' to a '0'. This returns the circuit back to its initial state.

The sequence of events described above can be followed in the schematic netlist shown in Figure 2.5(b). The state graph of Figure 2.5(c) indicates the sequence of events in yet another form where unique state codes have been assigned to ensure a correct and repeatable sequence of events.

The complex gate equivalent circuit of Figure 2.5 is simply a symbol with inputs x and y on the left and the output z . The behavior of this complex gate equivalent circuit must reflect the behavior specified in the Petri net and also that shown in the state graph.

2.7 Summary

In this chapter, we have laid a foundation using semantics to define time Petri nets, netlists, and state graphs. We have presented a design flow for our work that begins with a timed Petri net and a netlist, derives a state graph, and uses a complex gate equivalent representation during hazard verification. Finally, we have illustrated this work with two examples.