Asynchronous Circuit Design. Chris J. Myers
Copyright © 2001 by John Wiley & Sons, Inc.
ISBNs: 0-471-41543-X (Hardback); 0-471-22414-6 (Electronic)

8 Verification

Prove all things; hold fast that which is good.
—New Testament, I Thessalonians

All technology should be assumed guilty until proven innocent.
—David Brower

No amount of experimentation can ever prove me right; a single experiment can prove me wrong.
—Albert Einstein

A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.
—Douglas Adams, *Mostly Harmless*

When writing a specification of a circuit, we are usually trying to accomplish certain goals. For example, we may want to be sure that the protocol never leads to a deadlock or that whenever there is a request, it is followed by an acknowledgment possibly in a bounded amount of time. In order to validate that a specification will lead to a circuit that achieves these goals, simulation can be used, but this cannot guarantee complete coverage. This chapter, therefore, describes methods to verify that specifications meet their goals under all permissible delay behaviors.

After designing a circuit using one of the methods described in the previous chapters, we check the circuit by simulating a number of important cases until we are confident that it is correct. Unfortunately, anything short of exhaustive simulation will not guarantee the correctness of our design. This is especially problematic in asynchronous design, where a hazard may manifest as a failure only under a very particular set of delays. Therefore, it is necessary to use *verification* to check if a circuit operates correctly under all the allowed combinations of delay. In this chapter we describe methods for verifying a circuit's correctness under all permissible delay behavior.

8.1 PROTOCOL VERIFICATION

In this section we describe a method for specifying and verifying whether a protocol or circuit has certain desired properties. This type of verification is often called *model checking.* To specify the desired behavior of a combinational circuit, one typically uses *propositional logic.* For sequential circuits, it is necessary to describe behavior of a circuit over time, so one must use a *temporal logic.* In this section we introduce *linear-time temporal logic* (LTL)..

8.1.1 Linear-Time Temporal Logic

A temporal logic is a propositional logic which has been extended with operators to reason about future states of a system. LTL is a *linear-time* temporal logic in which truth values are determined along paths of states. The set of LTL formulas can be described recursively as follows:

1. Any signal u is a LTL formula.
2. If f and g are LTL formulas, so are:
   (a) $\neg f$ (not)
   (b) $f \wedge g$ (and)
   (c) $\bigcirc f$ (*next state operator*)
   (d) $f \ \mathbf{U} \ g$ (*strong until operator*)

We can describe the set of all allowed sequences using a SG, $\langle S,\ \delta,\ \lambda_S\rangle$, as described in Chapter 4. We require that the state transition relation is *total.* In other words, if any state has no successor, that state is assumed to have itself as a successor. Recall that $\delta \subseteq S \times T \times S$ is the set of possible state transitions. If there exists a state s_i for which there does not exist any transition t_j and state s_k such that $(s_i, t_j, s_k) \in \delta$, add $(s_i, \$, s_i)$ into δ, where \$ is a sequencing transition.

The truth of a LTL formula f can be defined with respect to a state s_i (denoted $s_i \models f$). A signal u is true in a state if the signal u is labeled with a 1 in that state [i.e., $\lambda_S(s_i)(u) = 1$]. The formula $\neg f$ is true in a state s_i when f is false in that state. The formula $f \wedge g$ is true when both f and g are true in s_i. The formula $\bigcirc f$ is true in state s_i when f is true in all next states s_j

reachable in one transition. The formula $f \ \mathbf{U}\ g$ is true in a state s_i when in all allowed sequences starting with s_i, f is true until g becomes true. A state in which g becomes true must always be reached in every allowed sequence. For this reason, this is often called the *strong until operator.* This relation can be defined more formally as follows:

$$\begin{array}{rll}
s_i \models u & \text{iff} & \lambda_S(s_i)(u) = 1 \\
s_i \models \neg f & \text{iff} & s_i \not\models f \\
s_i \models f \wedge g & \text{iff} & s_i \models f \text{ and } s_i \models g \\
s_i \models \bigcirc f & \text{iff} & \text{for all states } s_j \text{ such that } (s_i, t, s_j) \in \delta \ . \ s_j \models f \\
s_i \models f \ \mathbf{U}\ g & \text{iff} & \text{for all allowed sequences } (s_i, s_{i+1}, \ldots), \\
 & & \exists j \ . \ j \geq i \wedge s_j \models g \wedge (\forall k \ . \ i \leq k < j \Rightarrow s_k \models f)
\end{array}$$

It is often convenient to use some abbreviations to express some common LTL formulas. For example, in propositional logic, the formula $f \vee g$ can be used to express $\neg((\neg f) \wedge (\neg g))$. We also often use $f \Rightarrow g$ to express $(\neg f) \vee g$. New temporal operators include $\Diamond f$, which is used to express that f will eventually become true in all allowed sequences starting in the current state. The formula $\Box f$ is used to say that f is always true in all allowed sequences starting in the current state. Finally, a *weak until operator* (denoted $f \ \mathbf{W}\ g$) is used to state that f remains true until a state is reached in which g is true, but g never needs to hold as long as f holds. These formulas can be defined in terms of the previously defined ones as follows:

$$\begin{array}{rcl}
\Diamond f & \equiv & \mathit{true}\ \mathbf{U}\ f \ (\mathit{eventually\ operator}) \\
\Box f & \equiv & \neg\Diamond(\neg f) \ (\mathit{always\ operator}) \\
f \ \mathbf{W}\ g & \equiv & (f \ \mathbf{U}\ g \vee \Box f \ (\mathit{weak\ until\ operator})
\end{array}$$

Example 8.1.1 Let's return to our favorite wine shop example and write some formulas for the desired properties of a passive/active wine shop. First, we should not raise *ack_wine* until *req_wine* goes high:

$$\Box(\neg \mathit{ack\_wine} \ \Rightarrow \ (\neg \mathit{ack\_wine}\ \mathbf{U}\ \mathit{req\_wine})) \qquad (8.1)$$

This property states that in all states in which *ack_wine* is low, it should stay low until *req_wine* goes high. We also require that once we set *ack_wine* high, it must stay high until *req_wine* goes low again:

$$\Box(\mathit{ack\_wine} \ \Rightarrow \ (\mathit{ack\_wine}\ \mathbf{U}\ \neg \mathit{req\_wine})) \qquad (8.2)$$

On the patron side, once the shop has set *req_patron* high, it must hold it high until *ack_patron* goes high:

$$\Box(\mathit{req\_patron} \ \Rightarrow \ (\mathit{req\_patron}\ \mathbf{U}\ \mathit{ack\_patron})) \qquad (8.3)$$

Once *req_patron* is set low, it must stay low until *ack_patron* goes low:

$$\Box(\neg \mathit{req\_patron} \ \Rightarrow \ (\neg \mathit{req\_patron}\ \mathbf{U}\ \neg \mathit{ack\_patron})) \qquad (8.4)$$

Another important property is once the request and acknowledge wires on either side go high, they must be reset again:

$$\Box((\mathit{req\_wine} \wedge \mathit{ack\_wine}) \Rightarrow \Diamond(\neg \mathit{req\_wine} \wedge \neg \mathit{ack\_wine})) \qquad (8.5)$$
$$\Box((\mathit{req\_patron} \wedge \mathit{ack\_patron}) \Rightarrow \Diamond(\neg \mathit{req\_patron} \wedge \neg \mathit{ack\_patron})) \qquad (8.6)$$

We also don't want the wine to stay on the shelf forever, so after each bottle arrives, the patron should be called.

$$\Box(\mathit{ack\_wine} \Rightarrow \Diamond \mathit{req\_patron}) \qquad (8.7)$$

Finally, the patron should not arrive expecting wine in the shop before the wine has actually arrived.

$$\Box(\neg \mathit{ack\_patron} \Rightarrow (\neg \mathit{ack\_patron}\ \mathbf{U}\ \mathit{ack\_wine})) \qquad (8.8)$$

We can check whether or not a LTL formula is valid for a SG through a simple analysis. We can consider an LTL formula as being composed of propositional and temporal terms. Our first step is to mark each state that satisfies each of the propositional terms of the formula. Next, for each term of the form $\bigcirc f$ where f is a propositional formula, we mark each state in which f is true in all successors as satisfying $\bigcirc f$. To address terms of the form $f\ \mathbf{U}\ g$ requires a two-step procedure. First, we mark each state that satisfies g as also satisfying $f\ \mathbf{U}\ g$. Second, we mark each predecessor of a state marked as satisfying $f\ \mathbf{U}\ g$ that also satisfies f as satisfying $f\ \mathbf{U}\ g$. We then recursively mark the predecessors of these states until there is no change. Other temporal operators can be converted to their basic form for treatment. Now, to check the entire formula, we apply these procedures recursively to build up the formula.

Example 8.1.2 Let's check the validity of LTL formula 8.8 on a couple of potential protocols. The SG for the first is shown in Figure 8.1. This formula is trivially satisfied in the states in which *ack_patron* is high: $\{\langle F111\rangle, \langle 01F1\rangle, \langle R10F\rangle, \langle 110F\rangle, \langle RF00\rangle, \langle 1F00\rangle\}$. In the remaining states, we must verify that the formula $(\neg \mathit{ack\_patron}\ \mathbf{U}\ \mathit{ack\_wine})$ holds. To check this, we first mark the states where *ack_wine* is true, since this formula holds in these states (i.e., $\langle F01R\rangle$, $\langle 001R\rangle$, $\langle FR11\rangle$, $\langle 0R11\rangle$, $\langle F111\rangle$, and $\langle 01F1\rangle$). We then search for states which can reach the states we just marked in one step, and we check that in that state *ack_patron* is low. There is one such state, $\langle 10R0\rangle$, where indeed *ack_patron* is low, so we can mark that it satisfies the formula. We then continue our search, and we find that state $\langle R000\rangle$ reaches $\langle 10R0\rangle$ and has *ack_patron* low, so it also satisfies the formula. We have now marked all states as satisfying the formula, so the LTL formula is valid for this protocol. It can quickly be verified that this protocol satisfies all the other desired properties as well.

An SG for an alternative protocol is shown in Figure 8.2. We again begin by marking states with *ack_patron* high as trivially satisfying this formula (i.e., $\langle 11R1\rangle$, $\langle F111\rangle$, $\langle 011F\rangle$, $\langle 0FF0\rangle$, $\langle RF00\rangle$, and $\langle 1F00\rangle$).

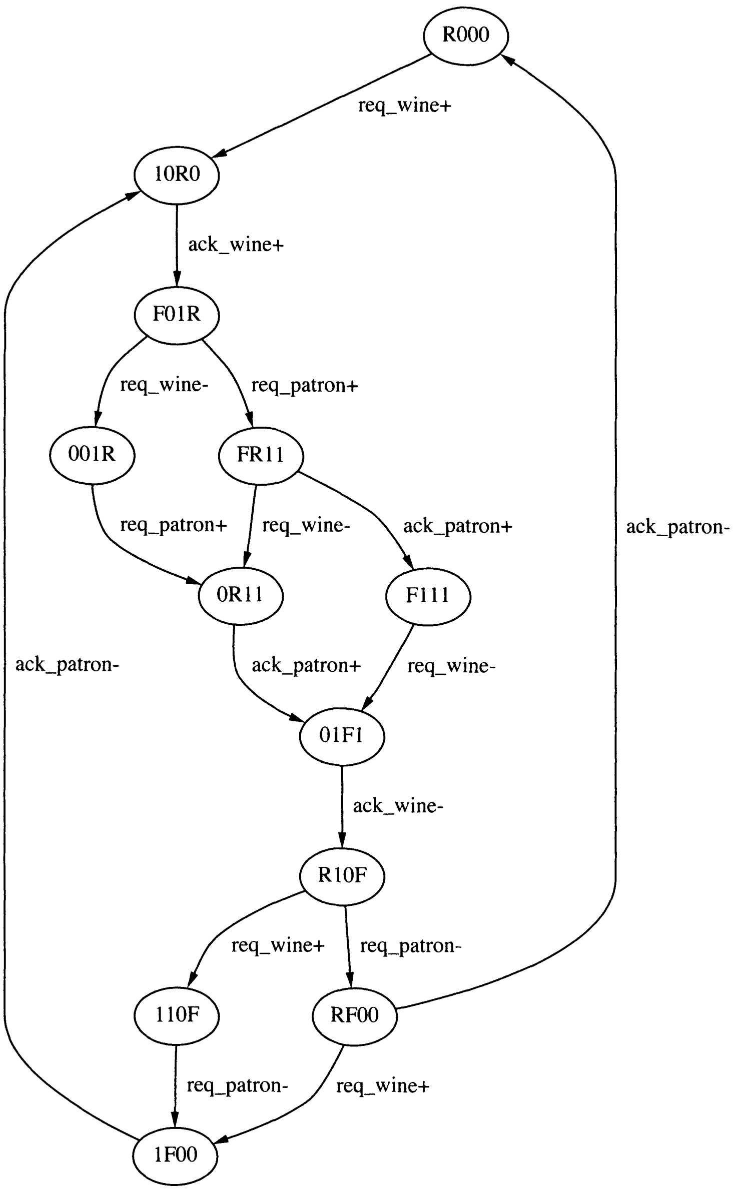


Fig. 8.1 SG for legal protocol for the passive/active shop (state vector is ⟨*req_wine*, *ack_patron*, *ack_wine*, *req_patron*⟩).

Again, we are left to check that in the remaining states the formula ($\neg$*ack_patron* **U** *ack_wine*) holds. We start by marking the states where *ack_wine* is true (i.e., $\langle FR11\rangle$, $\langle 0R11\rangle$, $\langle F111\rangle$, $\langle 011F\rangle$, $\langle 0FF0\rangle$, and $\langle 00F0\rangle$). Next, we look for states which can reach this set of states in one step, and we find $\langle 11R1\rangle$. In this state, *ack_patron* is high, so ($\neg$*ack_patron* **U** *ack_wine*) does not hold in this state. This in itself is not bad, since the entire formula does hold in this state. Consider, however, state $\langle 1RR1\rangle$, in which *ack_patron* is low. This state can reach $\langle 11R1\rangle$, so it also does not satisfy ($\neg$*ack_patron* **U** *ack_wine*). In this case, we have found a state in which the entire formula does not hold, so the LTL formula is not valid for this protocol. A sequence of events which leads to this failure would be

req_wine+, *req_patron*+, *ack_patron*+

Another alternative protocol is shown in Figure 8.3. A careful analysis determines that the patron will never arrive at a shop that has no wine. Unfortunately, the LTL formula that we used to express this property does not hold for this protocol. Consider state $\langle 1R01\rangle$, in which *ack_patron* is low. According to the formula, *ack_patron* must remain low until *ack_wine* goes high. However, the state $\langle 110F\rangle$ can be reached next, in which *ack_patron* is high. This violates the formula. This is okay, though, since the last bottle of wine is still sitting on the shelf. In fact, any protocol that allows *ack_wine* to reset before the patron receives the wine will violate this formula.

It is quite difficult (if not impossible) to find an LTL formula which admits only valid protocols and all valid protocols. Even if such a formula exists, how does one know when they found it? This is the key difficulty with model checking in that when verification succeeds, it does not mean that the protocol is correct but only that it has the property being checked.

8.1.2 Time-Quantified Requirements

The LTL formula $\Diamond f$ states that eventually f becomes true, but it puts no guarantee on how long before f will become true. If we wish to express properties such as *bounded response time*, it is necessary to extend the temporal logic that we use to specify timing bounds. One possible way of doing this is to annotate each temporal operator with a timing constraint. For example, $\Diamond_{<5} f$ states that f becomes true in less than 5 time units.

The set of *timed LTL* formulas can be described recursively as follows:

1. Any signal u is a timed LTL formula.
2. If f and g are timed LTL formulas, so are:
   (a) $\neg f$ (not)
   (b) $f \wedge g$ (and)
   (c) $f \ \mathbf{U}_{\sim c}\ g$ (*timed until operator*)

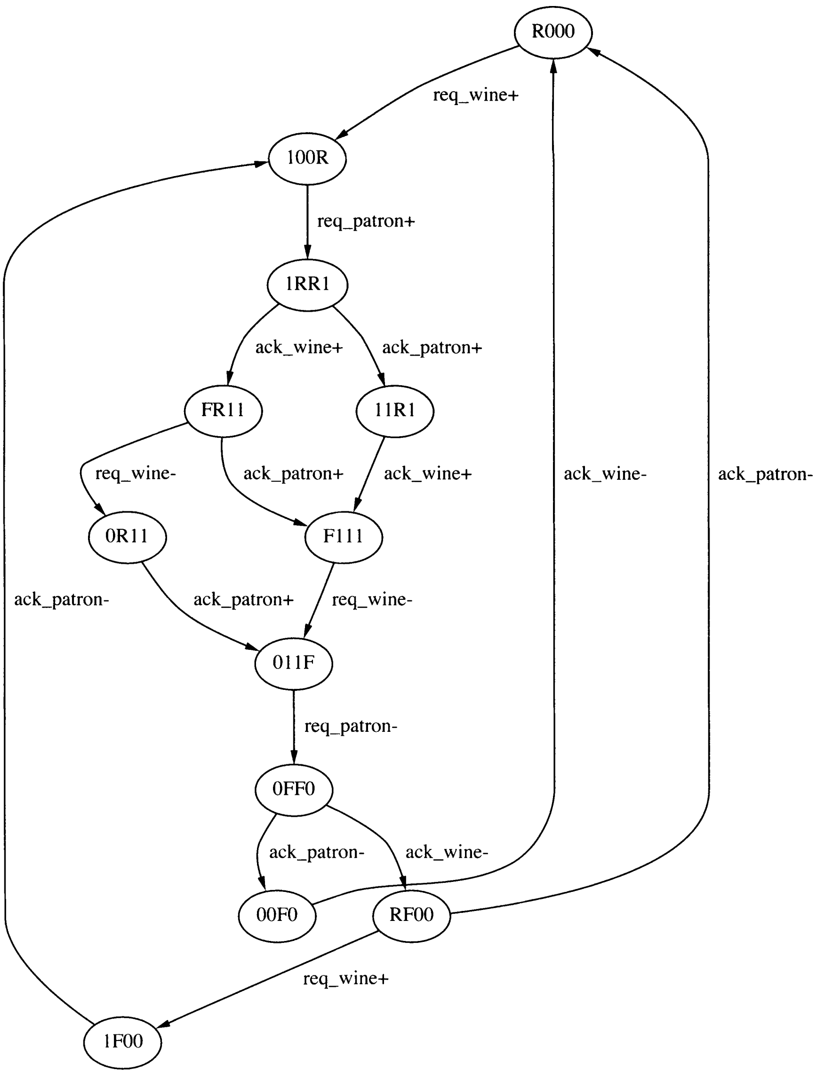


Fig. 8.2 SG for illegal protocol for the passive/active shop (state vector is $\langle$*req_wine*, *ack_patron*, *ack_wine*, *req_patron*$\rangle$).

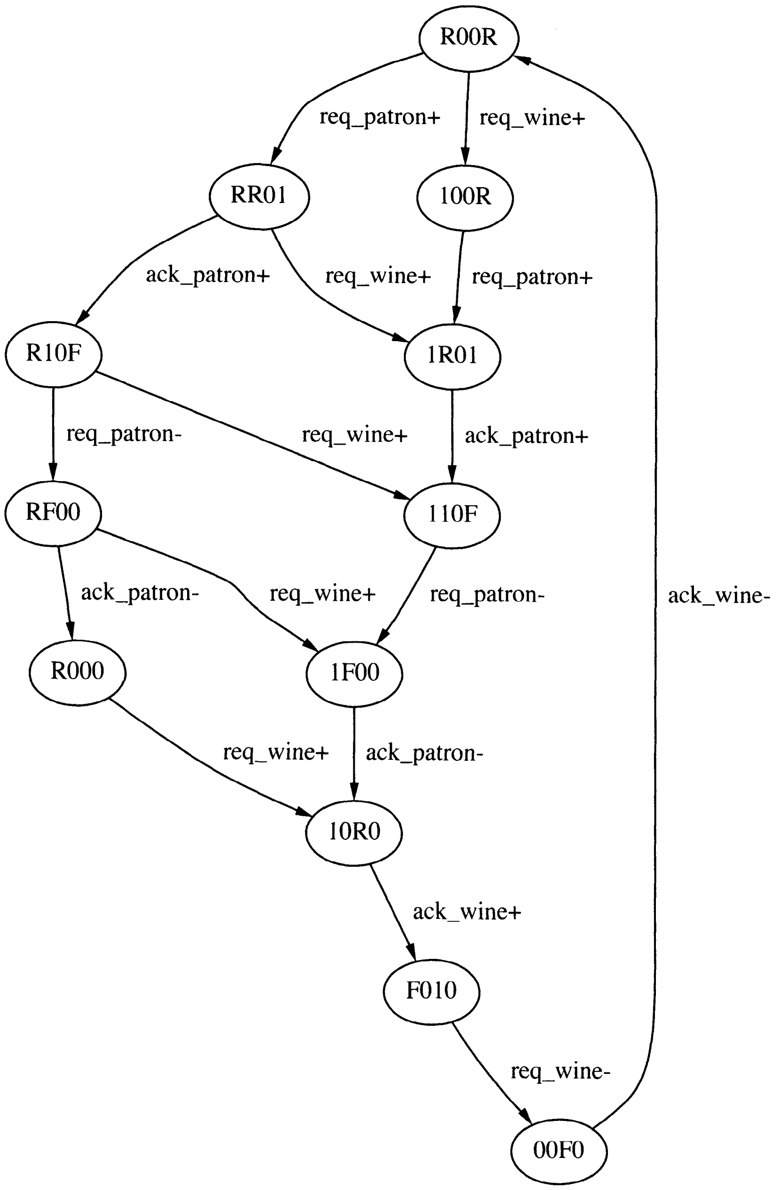


Fig. 8.3 SG for legal protocol for the passive/active shop that violates the property (state vector is ⟨*req_wine*, *ack_patron*, *ack_wine*, *req_patron*⟩).

where $\sim$ is $<$, $\leq$, $=$, $\geq$, $>$. Note that there is no next time operator, since when time is dense, there can be no unique next time.

Again, in timed LTL, we have the following set of abbreviations:

$$\begin{aligned}\Diamond_{\sim c} f &\equiv \mathit{true}\ \mathbf{U}_{\sim c}\ f\ (\mathit{timed\ eventually\ operator})\\ \Box_{\sim c} f &\equiv \neg\Diamond_{\sim c}(\neg f)\ (\mathit{timed\ always\ operator})\end{aligned}$$

Using the basic timed LTL primitives, we can also define temporal operators subscripted with time intervals.

$$\Diamond_{(a,b)} f \equiv \Diamond_{=a}\Diamond_{<(b-a)} f$$

Example 8.1.3 Let's again consider the passive/active wine shop and specify a few bounded response time properties. First, once the request and acknowledge wires on either side go high, they must be reset again within 10 minutes:

$$\Box((\mathit{req\_wine} \wedge \mathit{ack\_wine}) \Rightarrow \Diamond_{\leq 10}\ (\neg \mathit{req\_wine} \wedge \neg \mathit{ack\_wine}))$$

$$\Box((\mathit{req\_patron} \wedge \mathit{ack\_patron}) \Rightarrow \Diamond_{\leq 10}\ (\neg \mathit{req\_patron} \wedge \neg \mathit{ack\_patron}))$$

We also don't want the wine to age too long on the shelf, so after each bottle arrives, the patron should be called within 5 minutes:

$$\Box(\mathit{ack\_wine} \Rightarrow \Diamond_{\leq 5}\ \mathit{req\_patron})$$

In order to check a timed LTL property, the timed state space exploration algorithms described in Chapter 7 must be used. For example, to check the formula given above using the discrete-time analysis method, one can simply analyze the timed state space after it has been found. In particular, one could search for any state in which *ack_wine* is high and *req_patron* is low, and there must not exist any path from that state which takes 5 time steps before *req_patron* goes high. Using region- or zone-based analysis methods to check this property is a bit more involved and beyond the scope of this chapter.

8.2 CIRCUIT VERIFICATION

While model checking can be used for circuit verification, it has been found to be a bit unwieldy. In this section we present an alternative verification methodology based on *trace theory* for verifying whether a circuit implements a given specification.

8.2.1 Trace Structures

In order to verify that a circuit implements, or *conforms* to, a specification, it is necessary to check that all the possible behaviors of the circuit are allowed behaviors in the specification. To define these behaviors, we will use *traces*

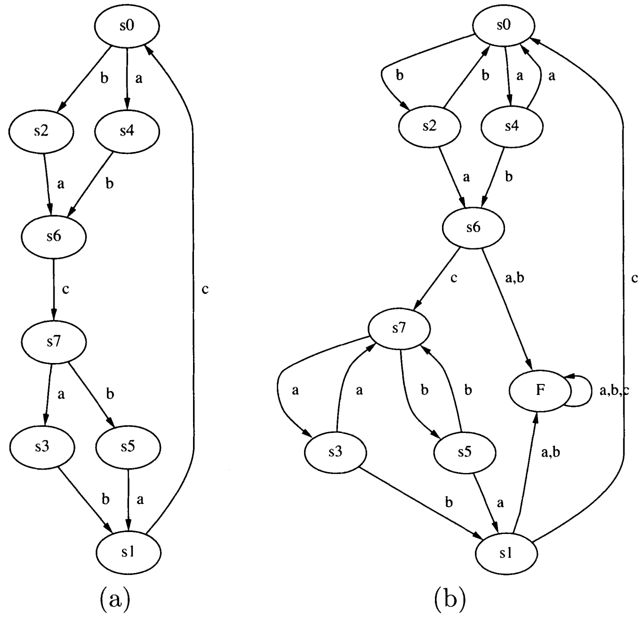


Fig. 8.4 (a) SG for a C-element. (b) Receptive SG for a C-element.

of events on signals. A trace is similar to an allowed sequence described earlier, but rather than keeping track of the states that the system has passed through, we track the signals which have had events on them.

Example 8.2.1 Consider the SG for a C-element in Figure 8.4(a). In this SG, one possible allowed sequence is (s_0, s_4, s_6, s_7, ...). The corresponding trace would be (a, b, c, ...).

The set of all traces is represented using a *trace structure.* In order to verify that a circuit is hazard-free, we will use a class of trace structures called *prefix-closed trace structures.* A prefix-closed trace structure can be described using a four-tuple: $\langle I, O, S, F \rangle$. The set I is the set of input signals, those signals controlled by the environment. The set O is the set of output signals, those controlled by the circuit. The set S is all traces which are considered successful. The set F is all traces which are considered a failure. At times, we may use A to indicate the set of all possible events and P to indicate the set of all possible traces. A trace structure must be *receptive.* In other words, the state of a circuit cannot prevent an input from happening (i.e., $PI \subseteq P$).

Example 8.2.2 If the signals a and b are inputs in the SG shown in Figure 8.4(a), the SG is not receptive since, for example, a is not allowed to occur in states s_4, s_6, s_3, and s_1. A receptive SG is shown in Figure 8.4(b). Note that now we must add a failure state, F, since if c is enabled to change and one of the inputs change instead, the C-element is hazardous and could glitch.

8.2.2 Composition

Due to the receptiveness requirement, if we wish to compose two circuits together and determine the corresponding trace structure, we must first make their signal sets match. Consider composing two trace structures: $T_1 = \langle I_1, O_1, S_1, F_1 \rangle$ and $T_2 = \langle I_2, O_2, S_2, F_2 \rangle$. If N is the set of signals in A_2 which are not in A_1, we must add N to I_1 and extend S_1 and F_1 to allow events on signals in N to occur at any time. Similarly, we must extend T_2 with those signals in A_1 but not in A_2. The function to perform this signal extension is called *inverse delete*, denoted $\mathrm{del}(N)^{-1}(X)$, where N is a set of signals and X is a set of traces. This function inserts elements of N^* between any consecutive signals in a trace in X (where N^* indicates zero or more events on signals in N). This function can be extended to a trace structure as follows:

$$\mathrm{del}(N)^{-1}(T) \quad = \quad \langle I \cup N, O, \mathrm{del}(N)^{-1}(S), \mathrm{del}(N)^{-1}(F) \rangle$$

Given two trace structures with *consistent signal sets* (i.e., $A_1 = A_2$ and $O_1 \cap O_2 = \emptyset$), they can be intersected as follows:

$$T_1 \cap T_2 \quad = \quad \langle I_1 \cap I_2, O_1 \cup O_2, S_1 \cap S_2, (F_1 \cap P_2) \cup (F_2 \cap P_1) \rangle$$

The implication of this definition is that a trace is considered a success in the composite only when it is a success in both circuits. It is considered a failure when it is a failure in either of the original circuits. One last thing to note is that the set of possible traces may be reduced through composition, since the set of possible traces in the composite is $P_1 \cap P_2$.

Using this definition and that of inverse deletion, composition can be defined as follows:

$$T_1 || T_2 \quad = \quad \mathit{del}(A_2 - A_1)^{-1}(T_1) \cap \mathit{del}(A_1 - A_2)^{-1}(T_2)$$

Example 8.2.3 Let us compose the trace structure for the C-element shown in Figure 8.4(b) with the trace structures for two inverters to form the circuit shown in Figure 8.5(a). The trace structure for an inverter is shown in Figure 8.6(a). The first thing we must do is introduce the notion of *renaming*. The trace structure for the inverter shown in Figure 8.6(a) has x as an input and y as an output. To make the signals match those in the diagram for the first inverter, we must rename x to c and y to a. We must also extend the trace structure to include the signal b as an additional input using the inverse delete function before it can be composed with the C-element. The result is shown in Figure 8.6(b).

The trace structure for the inverter shown in Figure 8.6(b) can now be composed with the trace structure for the C-element, and the result is shown in Figure 8.7. Note that the set of input signals, I, is now only b, while the set of output signals, O, is $\{a, c\}$. The failure state from the inverter can no longer be reached after composition, since a second c change cannot come before an a change. Also, sequences that involve two a changes in a row have also been removed.

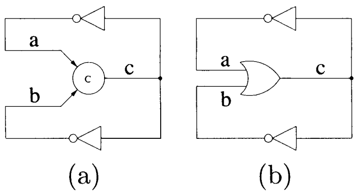


Fig. 8.5 (a) Simple C-element circuit. (b) Simple OR gate circuit.

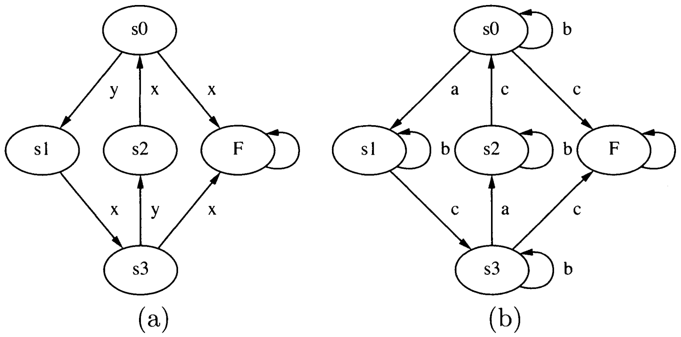


Fig. 8.6 (a) Receptive SG for an inverter. (b) SG for an inverter with input c, output a, and unconnected input b.

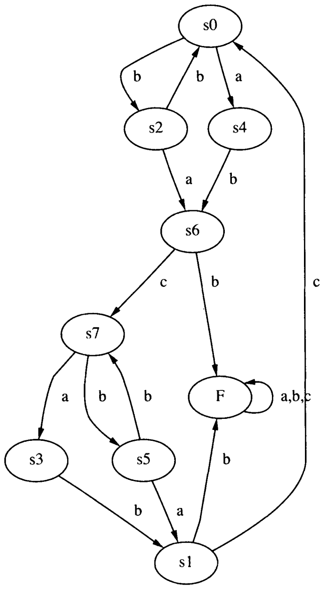


Fig. 8.7 (b) SG after composing one inverter with the C-element.

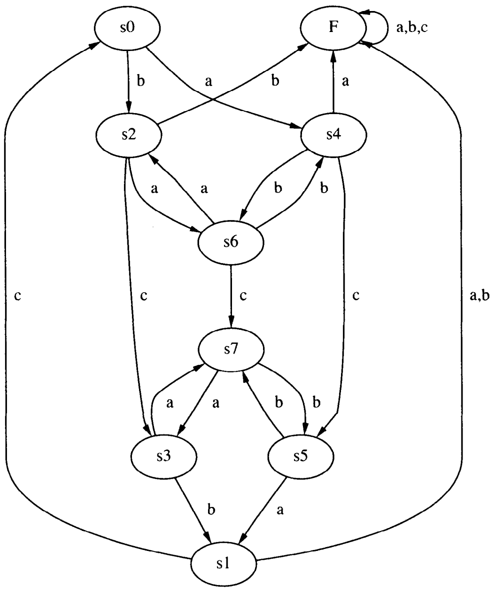


Fig. 8.8 Receptive SG for an OR gate.

If we rename and extend the trace structure in a similar fashion to create the second inverter and compose it with the trace structure shown in Figure 8.7, we get the trace structure shown in Figure 8.4(a). Notice that the failure state is no longer reachable. In other words, in this trace structure the failure set, F, is empty. Therefore, the circuit shown in Figure 8.5(a) is *failure-free.*

Example 8.2.4 Consider the composition of the trace structure for an OR gate shown in Figure 8.8 with the trace structure for two inverters [see Figure 8.6(a)] to form the circuit shown in Figure 8.5(b). First, we again do renaming to obtain the trace structure for the inverter shown in Figure 8.6(b). The trace structure for the inverter shown in Figure 8.6(b) can now be composed with the trace structure for the OR gate, and the result is shown in Figure 8.9(a). Note that the failure state can now be reached starting in state $s0$ after a b and c change, since the inverter driving signal a can become disabled.

If we rename and extend the trace structure in a similar fashion to create the second inverter and compose it with the trace structure shown in Figure 8.9(a), we get the trace structure shown in Figure 8.9(b). Notice that the failure state is still reachable. Starting in state $s0$, the failure state can be reached either by an a change followed by a c change or a b change followed by a c change. In each case, one of the inverters becomes disabled, causing a hazard. Note that this circuit is actually speed-independent as defined in Chapter 6, but it is not semi-modular.

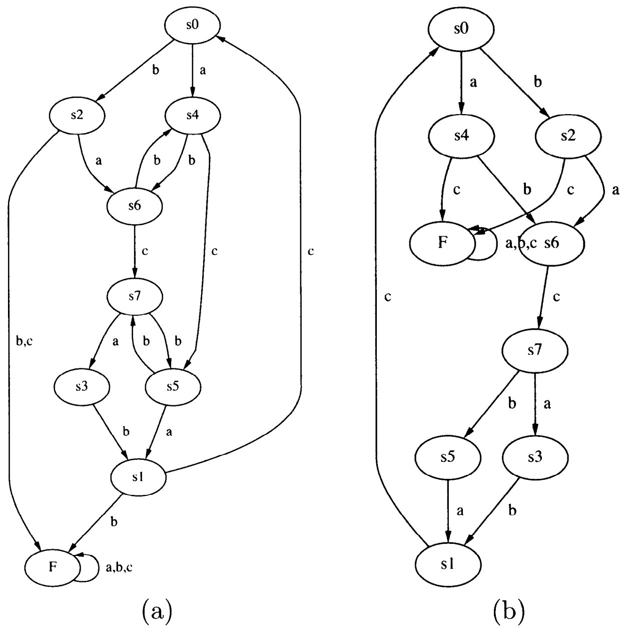


Fig. 8.9 (a) SG after composing one inverter with the OR gate. (b) SG after composing both inverters with the OR gate.

8.2.3 Canonical Trace Structures

To verify that a circuit correctly implements a specification, we can determine a trace structure for the circuit, T_I, and another for the specification, T_S, and show that T_I *conforms to* T_S (denoted $T_I \preceq T_S$). In other words, we wish to show that in any *environment* where the specification is failure-free, the circuit is also failure-free. An environment can be modeled by a trace structure, T_E, that has complementary inputs and outputs (i.e., $I_E = O_I = O_S$ and $O_E = I_I = I_S$). To check conformance, it is necessary to show that for every possible T_E, if $T_E \cap T_S$ is failure-free, so is $T_E \cap T_I$.

One important advantage of checking conformance is that it allows for *hierarchical verification.* In other words, once we have shown that an implementation conforms to a specification, we can use the specification in place of the implementation in verifying a larger system that has this circuit as a component. If there are many internal signals, this can be a huge benefit.

We say that two trace structures T_1 and T_2 are *conformation equivalent* (denoted $T_1 \sim_C T_2$) when $T_1 \preceq T_2$ and $T_2 \preceq T_1$. Unfortunately, if $T_1 \sim_C T_2$, it does not imply that $T_1 = T_2$. To make this true, we reduce prefix-closed trace structures to a canonical form using two transformations.

The first transformation is *autofailure manifestation.*

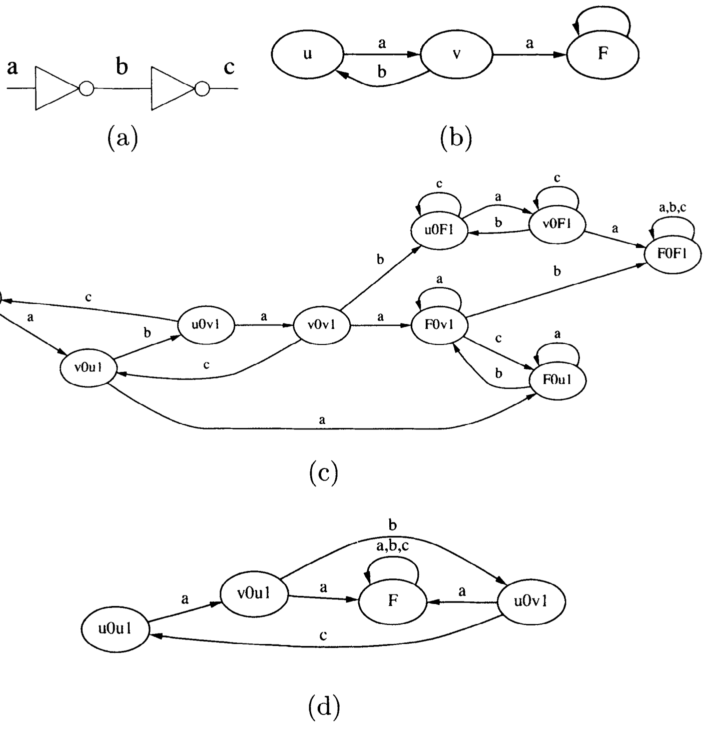


Fig. 8.10 (a) SG for an inverter. (b) Two inverters in series. (c) SG for two inverters in series. (d) SG for two inverters in series after simplification.

Example 8.2.5 Consider composing two inverters in series as in Figure 8.10(a). The state graph for a single inverter is shown in Figure 8.10(b). To simplify the explanation, we have reduced it from the five states in Figure 8.6(a) to 3 by folding states s_1 and s_2 to form state u and states s_0 and s_3 to form state v. After composing the two inverters, we find the SG shown in Figure 8.10(c). Any trace that ends in a state that includes F_0 or F_1 in its label is a failure trace. All other traces are considered successes.

Consider the partial trace (a, b, a), which ends in state $v0v1$. While this trace is successful, a possible next event is a transition on the output signal b which would lead to a failure. In this state, there is no way the environment can prevent this failure from happening. While it is possible that the output c could happen first, there is no guarantee that this will happen. In verification, any circuit which has a potential for a failure should be considered as failing verification. This type of failure is called an *autofailure*, since the circuit itself causes the failure. Autofailure manifestation adds the trace (a, b, a) to the failure set.

More formally, an autofailure is a trace x which can be extended by a signal $y \in O$ such that $xy \in F$. Another way of denoting this is $F/O \subseteq F$, where F/O is defined to be $\{x \mid \exists y \in O \,.\, xy \in F\}$. We also add to the failure set any trace that has a failure as a prefix (i.e., $FA \subseteq F$). The result of these two changes (assuming that $S \neq \emptyset$) is that any failure trace has a prefix that is a success, and the signal transition that causes it to become a failure is on an input signal. In other words, the circuit fails only if the environment sends a signal change that the circuit is not prepared for, and in this case, the circuit is said to *choke*.

The second transformation is called *failure exclusion*. In this transformation, we make the success and failure sets disjoint. When a trace occurs in both, it means that the circuit may or may not fail, but this again indicates a dangerous circuit. Therefore, we remove from the success set any trace which is also a failure (i.e., $S = S - F$).

Example 8.2.6 After applying both transformations, our simplified SG for two inverters in series is shown in Figure 8.10(d).

A *canonical prefix-closed trace structure* is one which satisfies the following three requirements:

1. Autofailures are failures (i.e., $F/O \subseteq F$).
2. Once a trace fails, it remains a failure (i.e., $FA \subseteq F$).
3. No trace is both a success and a failure (i.e., $S \cap F = \emptyset$).

In a canonical prefix-closed trace structure, the failure set is not necessary, so it can be represented with the triple $T = \langle I, O, S \rangle$. We can determine the failure set as follows:

$$F = [(SI \cup \{\epsilon\}) - S]A^*$$

In other words, any successful trace when extended with an input signal transition and is no longer found in the success set is a failure. Furthermore, any such failure trace can be extended indefinitely with other input or output signal transitions, and it will always be a failure.

8.2.4 Mirrors and Verification

To check conformance of a trace structure T_I to another T_S, we said that it is necessary to check that in all environments in which T_S is failure-free, T_I is also failure-free. It is difficult to imagine performing such a check. Fortunately, we can construct a unique worst-case environment that we can use to perform this feat in a single test. This environment is called a *mirror* of T (denoted T^M). If we have a canonical prefix-closed trace structure, the mirror can be constructed simply by swapping the inputs and outputs (i.e., $I^M = O$, $O^M = I$, and $S^M = S$).

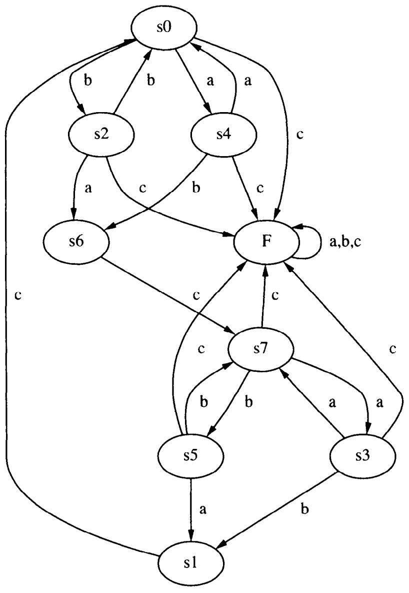


Fig. 8.11 Mirror for a C-element.

Example 8.2.7 The mirror of the C-element shown in Figure 8.4(b) is shown in Figure 8.11. Note that changing the inputs and outputs has the effect of changing the failure set. Recall that in canonical trace structures a failure occurs when an input happens at the wrong time. For the mirror of the C-element, the signal c is now the only input. If c changes in a state in which it was not expected to change, the trace is a failure. Also, note that changes on signals a and b are no longer allowed if they cause a failure.

Using the following theorem, we can use the mirror to check conformance of an implementation, T_I, to a specification, T_S.

Theorem 8.1 (Dill, 1989) *If* $T_I || T_S^M$ *is failure-free,* $T_I \preceq T_S$.

Example 8.2.8 Consider the merge element shown in Figure 8.12(a). A general merge accepts an input on either a or b and produces an output on c [see Figure 8.12(b)]. An alternating merge accepts an input on a and produces a c, then accepts an input on b and produces a c [see Figure 8.12(c)]. There are two questions we can ask. First, if we require an alternating merge, can we replace it with a general merge? Second if we require a general merge, can we replace it with an alternating merge?

To answer the first question, the SG shown in Figure 8.12(b) is our implementation, T_I, the one in Figure 8.12(c) is our specification, T_S, and we want to check if $T_I \preceq T_S$. To do this, we construct the mirror for

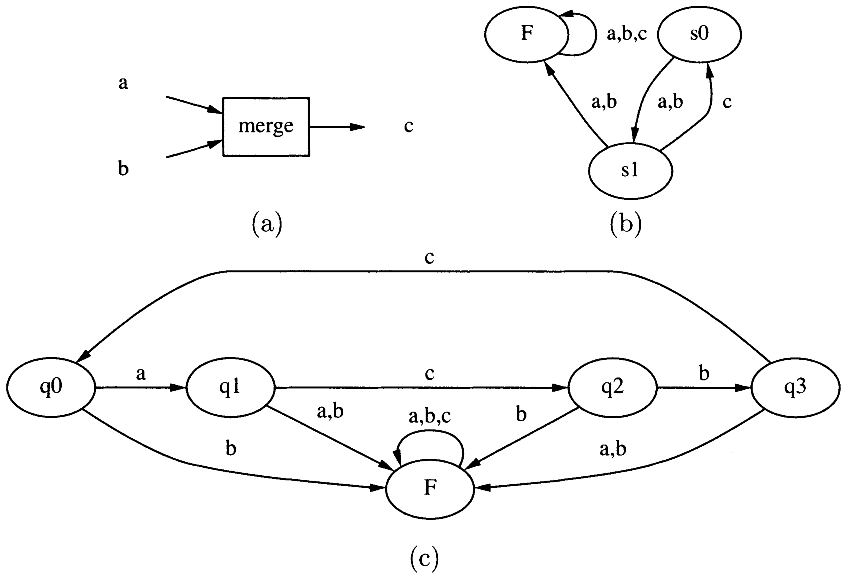


Fig. 8.12 (a) Merge element. (b) SG for a general merge element. (c) SG for an alternating merge.

T_S (i.e., the alternating merge), which is shown in Figure 8.13(a). We then compose this trace structure with the one for the general merge shown in Figure 8.12(b). The result is shown in Figure 8.13(b), and it is failure-free. This means that when an alternating merge is required, a general merge is a *safe substitute.*

To answer the second question, we must construct the mirror for the general merge, which is shown in Figure 8.13(c). We compose this trace structure with the one for the alternating merge shown in Figure 8.12(c), and the result is shown in Figure 8.13(d). This trace structure is not failure-free, so it is not safe to substitute an alternating merge when a general merge is required. Consider the initial state $q0s0$. In a general merge, it must be able to accept either an a or a b, while an alternating merge can only accept an a. If it receives a b, it would fail.

8.2.5 Strong Conformance

One limitation with this approach to verification is that it checks only safety properties. In other words, if a circuit verifies, it means that it does nothing bad. It does not mean, however, that it does anything good. For example, consider the trace structure for a "block of wood." A block of wood would accept any input, but it would never produce any output (i.e., $T = \langle I, O, I^* \rangle$). Assuming that the inputs and outputs are made to match, a block of wood would conform to any specification.

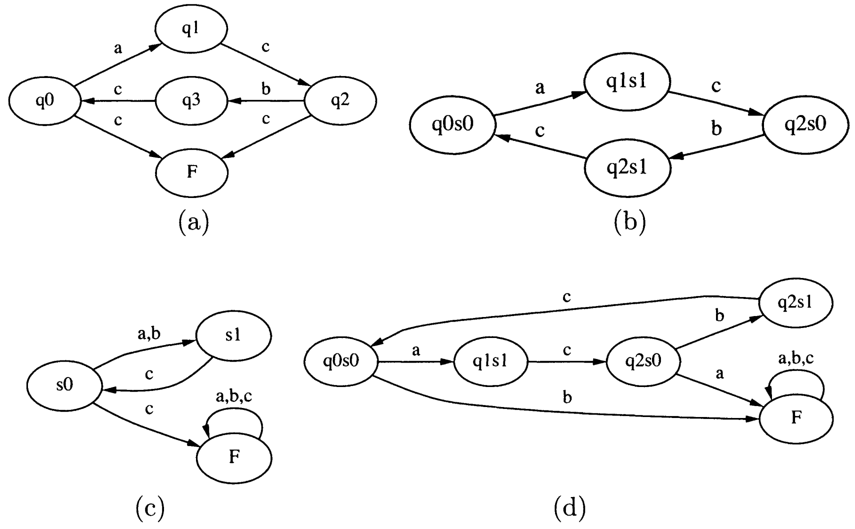


Fig. 8.13 (a) SG for mirror of alternating merge. (b) SG when checking if general merge conforms to alternating merge. (c) SG for mirror of general merge. (d) SG when checking if alternating merge conforms to general merge.

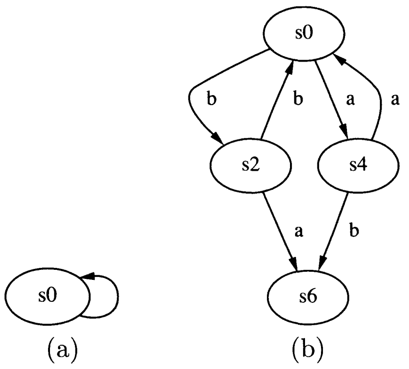


Fig. 8.14 (a) SG for a block of wood. (b) SG for a block of wood composed with the mirror of a C-element specification.

Example 8.2.9 Consider composing a block of wood whose behavior is shown in Figure 8.14(a) with the mirror of a C-element shown in Figure 8.11. The result is shown in Figure 8.14(b), and it is failure-free. Therefore, the block of wood conforms to the specification of a C-element. It is clear, however, that the block of wood is not a very good implementation of a C-element.

The notion of *strong conformance* removes this problem. T_1 *conforms strongly to* T_2 (denoted $T_1 \sqsubseteq T_2$) if $T_1 \preceq T_2$ and $S_1 \supseteq S_2$. In other words, all successful traces of T_2 must be included in the successful traces of T_1.

Example 8.2.10 Consider again the block-of-wood implementation of a C-element. The C-element has a successful trace (a, b, c), which is not in the set of successful traces of the block of wood. Therefore, the block of wood does not strongly conform to the specification of a C-element.

8.2.6 Timed Trace Theory

A *timed trace* is a sequence of $x = (x_1, x_2, \ldots)$ where each x_i is an event/time pair of the form (e_i, τ_i) such that:

- $e_i \in A$, where A is the set of signals.
- $\tau_i \in \mathbf{Q}$, where $\mathbf{Q}$ is the set of nonnegative rational numbers.

A timed trace must satisfy the following two properties:

- *Monotonicity*: for all i, $\tau_i \leq \tau_{i+1}$.
- *Progress*: if x is infinite, then for every $\tau \in \mathbf{Q}$ there exists an index i such that $\tau_i > \tau$.

Given a module M defined by a trace structure $\langle I, O, S \rangle$ and a trace $x \in S$, we say that the module M allows time to advance to time τ if for each $w' \in I \cup O$ and $\tau' < \tau$ such that $x(w', \tau') \in S$ implies that $x(w', \tau'') \in S$ for some $\tau'' \geq \tau$. Intuitively, this means that after trace x has happened, module M can allow time to advance to τ without needing an input or producing an output. We denote this by the predicate *advance_time*(M, x, τ).

Recall that we defined a failure to mean that some module produces an output when some other module is not ready to receive this as an input. In the timed case, this is complicated further by the fact that it must also be checked that the output is produced at an acceptable time. Consider a module $M = \langle I, O, S \rangle$ composed of several modules $\{M_1, \ldots, M_n\}$, where $M_k = \langle I_k, O_k, S_k \rangle$. Consider also a timed trace $x = (x_1, \ldots, x_m)$, where $x_m = (w, \tau)$ and $w \in O_k$ for some $k \leq n$. This trace causes a failure if *advance_time*$(M, (x_1, \ldots, x_{m-1}), \tau)$, $x \in S_k$, but $x \notin S$. Intuitively, this means that some module produces a transition on one of its outputs before some module is prepared to receive it. These types of failures are called *safety failures.*

A *timing failure* occurs when some module does not receive an input in time. In other words, either some input fails to occur or occurs later than required. There are potentially several ways to characterize timing failures formally, with each choice having different effects on the difficulty of verification. In particular, for the most general definition, it is no longer possible to use mirrors without some extra complexity which is beyond the scope of this chapter.

To verify a timed system, we must use one of the timed state space exploration algorithms described in Chapter 7. Let's consider using the discrete-time analysis method to find the timed state space for the implementation and the mirror of the specification. We can again compose them and check if the failure state is reachable. The application of region- or zone-based analysis methods for timing verification are a bit more involved and beyond the scope of this chapter.

8.3 SOURCES

Logics to represent time have long been discussed in philosophy circles. The modern temporal logics have their origin in the work of Kripke [212]. In recent years, it has seen increasing use in verification of both hardware and software systems [123, 303, 316]. The application of temporal logic to the verification of asynchronous circuits was proposed by Browne et al. [50] and Dill and Clarke [116]. A similar approach is taken by Berthet and Cerny using characteristic functions rather than a temporal logic [41]. Weih and Greenstreet have combined model checking with symbolic trajectory evaluation to verify speed-independent datapath circuits [398]. Lee et al. utilized a formalism similar to temporal logic called synchronized transitions to verify asynchronous circuits [228]. Bailey et al. utilized the Circal language and its associated algebra to verify asynchronous designs [21]. Adaptations of temporal logic to include timing have been developed by numerous people. One of the more famous ones is due to Abadi and Lamport [1]. Techniques for timed model checking of asynchronous circuits are described by Hamaguchi et al. [159], Burch [61], Yoneda and Schlingloff [415], and Vakilotojar et al. [386].

Trace theory was first applied to circuits by Snepscheut [361] and Udding [381] for the specification of delay-insensitive circuits. The application of trace theory to the verification of speed-independent circuits was pioneered by Dill [113, 115]. Section 8.2 follows Dill's work closely. An improved verifier was developed by Ebergen and Berks [117]. The notion of strong conformance was introduced by Gopalakrishnan et al. [151].

Recently, several techniques have been proposed to avoid a complete enumeration of the state space. These techniques utilize *partial orders* [149, 317, 416], *stubborn sets* [388], *unfoldings* [264, 266], or *cube approximations* [26, 28] to greatly reduce the size of the representation needed for the state space. Yoneda and Schlingloff developed a partial order method for the verification of timed systems [415].

Timed trace theory and a verification method based on discrete time were described by Burch [62, 63]. Devadas developed a technique for verifying Huffman circuits using back-annotated bounded delay information [112]. Rokicki and Myers employed a zone-based technique to verify timed asynchronous circuits [287, 325, 326]. These techniques have since been improved and applied to numerous examples by Belluomini et al. [31, 32, 33, 34]. Semenov and Yakovlev have developed an unfolding technique to verify timed circuits modeled using time Petri nets [349]. Yoneda and Ryu extended their timed partial order method to timed trace theoretic verification [414]. The problems with conformance and mirroring in the verification of timed circuits are described in [425]. Recently, there has been some research that employs implicit or *relative timing* assumptions (i.e., the timing of some sequence of events is assumed to be larger than some other sequence) to verify timed circuits [292, 310, 366].

Problems

8.1 Linear-Time Temporal Logic

Consider the situation where there are two shops to which the winery can deliver wine. It communicates with the first using the wires *req_wine1* and *ack_wine1* and the second with *req_wine2* and *ack_wine2*. Write an LTL formula that says that the winery is *fair*. In other words, the winery will deliver wine to both shops.

8.2 Linear-Time Temporal Logic

Again consider the situation where there are two shops to which the winery can deliver wine. Write an LTL formula that says that the winery at some point decides to sell only to one shop in the future.

8.3 Linear-Time Temporal Logic

If either the winery can stop producing wine or the patron loses interest in buying wine, which of LTL formulas 8.1 to 8.8 would be violated? How could they be fixed?

8.4 Protocol Verification

Check if the SG in Figure 8.15(a) satisfies the following LTL formula:

$$\Box(y \Rightarrow (x \mathbf{U} z))$$

If not, indicate which states violate the formula.

8.5 Protocol Verification

Check if the SG in Figure 8.15(b) satisfies the following LTL formula:

$$\Box(y \Rightarrow (x \mathbf{U} z))$$

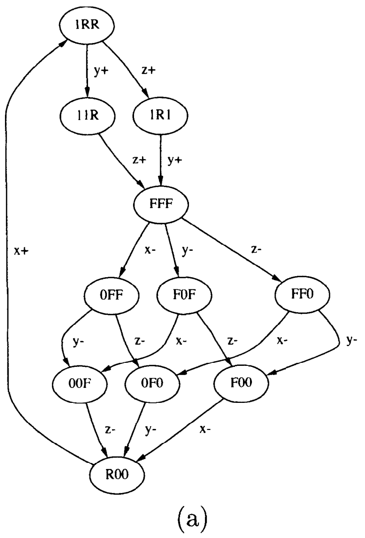
If not, indicate which states violate the formula.

8.6 Timed Linear-Time Temporal Logic

Again consider the situation where there are two shops to which the winery can deliver wine. Write a timed LTL formula that says that the winery is fair in bounded time. In other words, the winery will not stop delivering wine to one shop for more than 60 minutes.

8.7 Trace Structures

Give a receptive trace structure for a NAND gate.



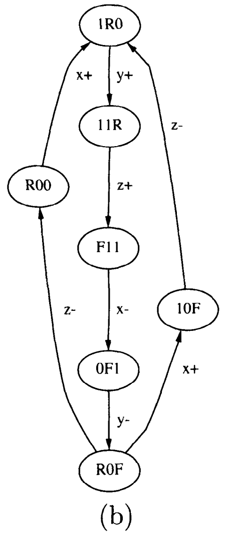


Fig. 8.15 SGs for Problems 8.4 and 8.5 (state vector $\langle x, y, z \rangle$).

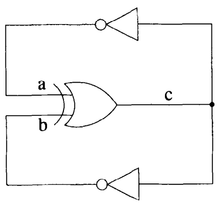


Fig. 8.16 Circuit for Problem 8.9.

8.8 Trace Structures

Give a receptive trace structure for an XOR gate.

8.9 Trace Structure Composition

Use composition of trace structures to determine whether or not the circuit shown in Figure 8.16 is failure-free.

8.10 Trace Structure Composition

Use composition of trace structures to determine whether or not the circuit shown in Figure 8.17 is failure-free.

8.11 Canonical Trace Structures

Transform the trace structure shown in Figure 8.18 to a canonical prefix-closed trace structure.

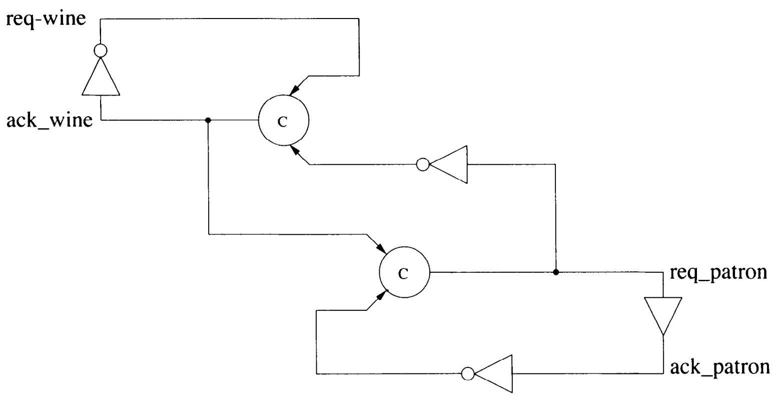


Fig. 8.17 Circuit for Problem 8.10.

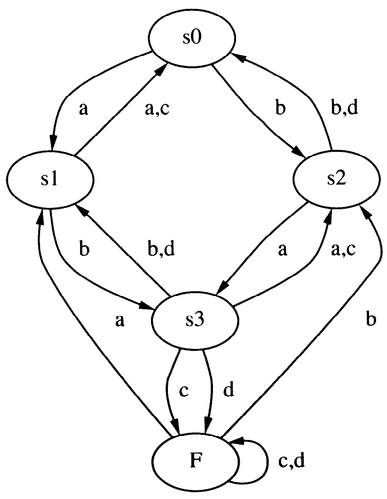


Fig. 8.18 Trace structure for Problem 8.11.

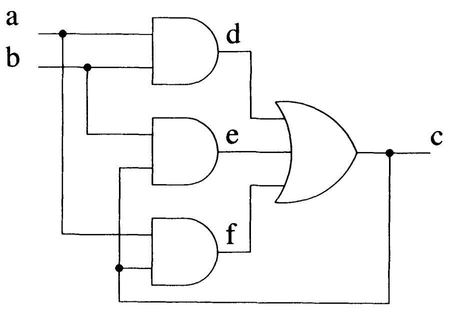


Fig. 8.19 Circuit for Problem 8.12.

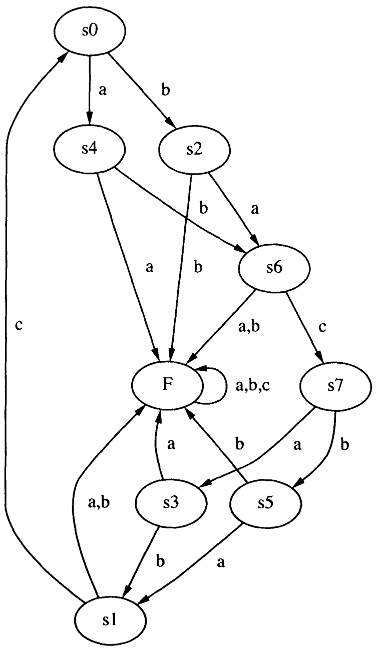


Fig. 8.20 Specification for Problems 8.12 and 8.13.

8.12 Mirrors and Verification

Use composition to create a trace structure for the circuit shown in Figure 8.19. Find the mirror of the specification for a C-element shown in Figure 8.20. Compose the trace structure for the circuit and the mirror of the specification to determine if the circuit conforms to the specification (i.e., the circuit is a correct implementation of a C-element). If the circuit does not conform, give a sequence of transitions that causes the circuit to fail.

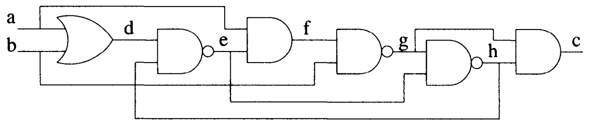


Fig. 8.21 Circuit for Problem 8.13.

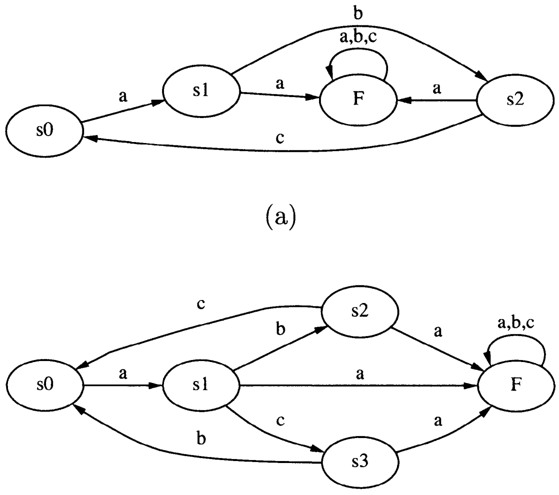


Fig. 8.22 (a) Sequencer and (b) fork.

8.13 Mirrors and Verification

Use composition to create a trace structure for the circuit shown in Figure 8.21. Find the mirror of the specification for a C-element shown in Figure 8.20. Compose the trace structure for the circuit and the mirror of the specification to determine if the circuit conforms to the specification (i.e., the circuit is a correct implementation of a C-element). If the circuit does not conform, give a sequence of transitions that causes the circuit to fail.

8.14 Strong Conformance

A sequencer receives an input a and generates an output b followed by another output c. The trace structure for a sequencer is shown in Figure 8.22(a). A fork receives an input a and generates outputs b and c in parallel. The trace structure for a fork is shown in Figure 8.22(b). Show that a sequencer conforms to a fork, but a fork does not conform to a sequencer. Does a sequencer strongly conform to a fork?