Asynchronous Circuit Design

Chris J. Myers

Lecture 8: Verification
Chapter 8

Protocol Verification

- Specification for circuit usually trys to accomplish certain goals.
- Examples:
  - Protocol never deadlocks.
  - Whenever there is a request, it is followed by an acknowledgement possibly in a bounded amount of time.
- Can check by simulating a number of important cases.
- Simulation does not guarantee correctness of the design.
- Big problem in asynchronous design where a problem only manifests under a very particular set of delays.
- Verification can also be used to check if a specification meets its goals under all permissable delay behaviors.

Model Checking

- *Model checking* is the process of verifying whether a protocol, circuit, or other type of system has certain desired properties.
- To specify desired behavior of a combinational circuit, one can use *propositional logic*.
- For sequential circuits, it is necessary to describe behavior of a circuit over time, so one must use a *propositional temporal logic*.
- *Linear-time temporal logic* (LTL) is presented here.

Linear-time Temporal Logic (LTL)

- A temporal logic is a propositional logic which has been extended with operators to reason about future states of a system.
- 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:
     1. $\neg f$ (not)
     2. $f \wedge g$ (and)
     3. $\bigcirc f$ (*next state operator*)
     4. $f \mathbf{U} g$ (*strong until operator*)

LTL Semantics

- Truth of formula f is defined with respect to a state s_i ($s_i \models f$).
- $\neg f$ is true in a state s_i when f is false in that state.
- $f \wedge g$ is true when both f and g are true in s_i.
- $\bigcirc f$ is true in state s_i when f is true in all next states s_j reachable in one transition.
- $f \mathbf{U} g$ is true in a state s_i when in all allowed sequences starting with s_i, f is true until g becomes true.

Formal LTL Semantics

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

LTL Abbreviations

- $\Diamond f$ means f will eventually become true in all allowed sequences starting in the current state.

$$\Diamond f \quad \equiv \quad true\ \mathbf{U}\ f$$

- $\Box f$ means f is always true in all allowed sequences.

$$\Box f \quad \equiv \quad \neg\Diamond(\neg f)$$

- $f\ \mathbf{W}\ g$ means f is always true or until g.

$$f\ \mathbf{W}\ g \quad \equiv \quad (f\ \mathbf{U}\ g) \vee \Box f$$

Desired Properties for a Passive/Active Wine Shop

- Should not raise *ack_wine* until *req_wine* goes high:

$$\Box(\neg ack_wine \Rightarrow (\neg ack_wine\ \mathbf{U}\ req_wine))$$

- Once *ack_wine* is high, it must stay high until *req_wine* goes low:

$$\Box(ack_wine \Rightarrow (ack_wine\ \mathbf{U}\ \neg req_wine))$$

- Once the shop has set *req_patron* high, it must hold it high until *ack_patron* goes high:

$$\Box(req_patron \Rightarrow (req_patron\ \mathbf{U}\ ack_patron))$$

- Once the shop sets *req_patron* low, it must hold it low until *ack_patron* goes low:

$$\Box(\neg req_patron \Rightarrow (\neg req_patron\ \mathbf{U}\ \neg ack_patron))$$

Desired Properties for a Passive/Active Wine Shop

- Once the request and acknowledge wires on either side go high, they must be reset again:

$$\Box((req_wine \wedge ack_wine) \Rightarrow \Diamond(\neg req_wine \wedge \neg ack_wine))$$
$$\Box((req_patron \wedge ack_patron) \Rightarrow \Diamond(\neg req_patron \wedge \neg ack_patron))$$

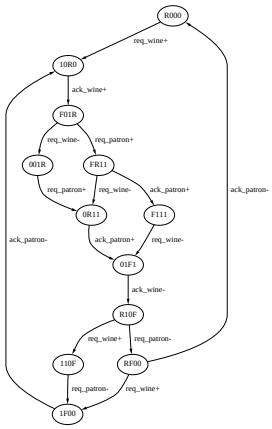
- The wine should not stay on the shelf forever, so after each bottle arrives, the patron should be called.

$$\Box(ack_wine \Rightarrow \Diamond req_patron)$$

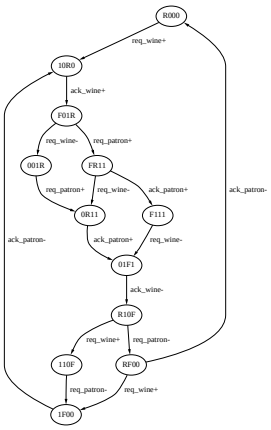
- The patron should not arrive expecting wine in the shop before the wine has actually arrived.

$$\Box(\neg ack_patron \Rightarrow (\neg ack_patron\ \mathbf{U}\ ack_wine))$$

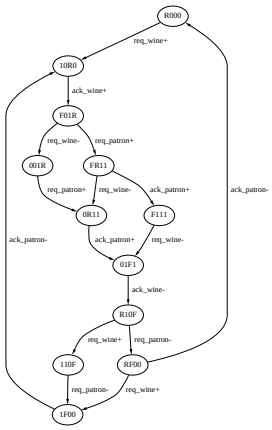
$\Box(\neg ack_wine \Rightarrow (\neg ack_wine\ \mathbf{U}\ req_wine))$



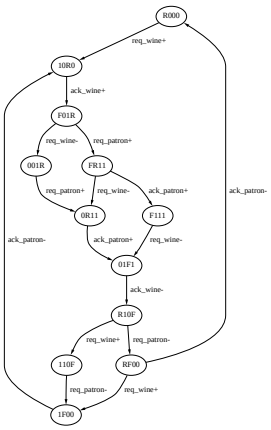
$\Box(ack_wine \Rightarrow (ack_wine\ \mathbf{U}\ \neg req_wine))$



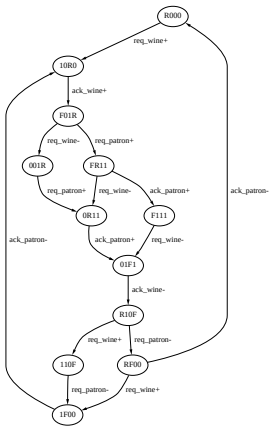
$\Box(req_patron \Rightarrow (req_patron\ \mathbf{U}\ ack_patron))$



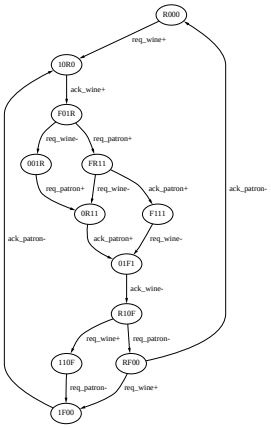
$\Box(\neg req_patron \Rightarrow (\neg req_patron \mathbf{U} \neg ack_patron))$



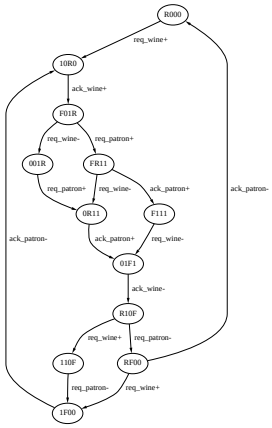
$\Box((req_wine \wedge ack_wine) \Rightarrow \Diamond(\neg req_wine \wedge \neg ack_wine))$



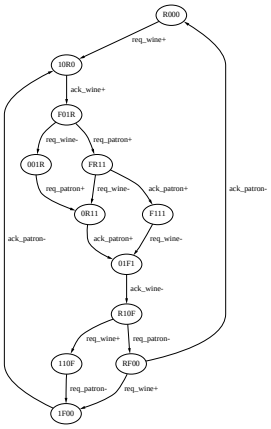
$\Box((req_patron \wedge ack_patron) \Rightarrow \Diamond(\neg req_patron \wedge \neg ack_patron))$



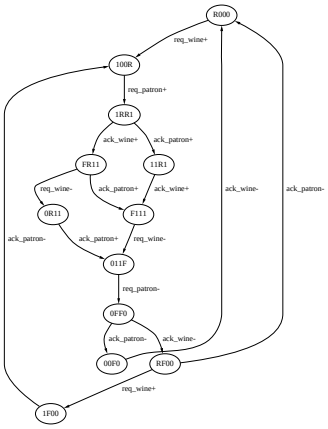
$\Box(ack_wine \Rightarrow \Diamond req_patron)$



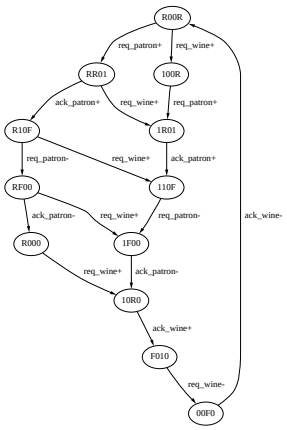
$\Box(\neg ack_patron \Rightarrow (\neg ack_patron \mathbf{U} ack_wine))$



$\Box(\neg ack_patron \Rightarrow (\neg ack_patron \mathbf{U} ack_wine))$



$\Box(\neg ack_patron \Rightarrow (\neg ack_patron \ \mathbf{U} \ ack_wine))$



Timed LTL

- $\Diamond f$ states that eventually f becomes true, but it puts no guarantee on how long before f will become true.
- To express *bounded response time*, it is necessary to extend the temporal logic that we use to specify timing bounds.
- In *timed LTL*, each temporal operator is annotated with a timing constraint.
- $\Diamond_{<5} f$ states that f becomes true in less than 5 time units.

Timed LTL Formulas

- Timed LTL formulas can be described recursively as follows:
  1. Any atomic proposition $p \in AP$ is a CTL formula.
  2. If f and g are CTL formulas then so are:
     1. $\neg f$ (not)
     2. $f \wedge g$ (and)
     3. $f \ \mathbf{U}_{\sim c} \ g$

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

Timed LTL Abbreviations

$$\Diamond_{\sim c} f \equiv true \ \mathbf{U}_{\sim c} \ f$$
$$\Box_{\sim c} f \equiv \neg \Diamond_{\sim c} (\neg f)$$

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

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

Some Bounded Response Time Properties

- Once the request and acknowledge wires on either side go high, they must be reset again within 10 minutes:

$$\Box((req_wine \wedge ack_wine) \Rightarrow$$
$$\Diamond_{\leq 10} (\neg req_wine \wedge \neg ack_wine))$$
$$\Box((req_patron \wedge ack_patron) \Rightarrow$$
$$\Diamond_{\leq 10} (\neg req_patron \wedge \neg ack_patron))$$

- 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:

$$\Box(ack_wine \Rightarrow \Diamond_{\leq 5} req_patron)$$

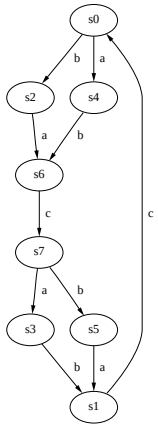
Circuit Verification

- Can check circuit by simulating a number of important cases.
- Simulation does not guarantee correctness of the design.
- Big problem in asynchronous design where a hazard may only manifest as a failure under a very particular set of delays.
- Verification checks if a circuit operates correctly under all the allowed combinations of delay.

Traces

- To verify a circuit *conforms* to a specification, it is necessary to check that all its behaviors are allowed by the specification.
- Define using *traces* of events on signals.
- A trace is similar to an allowed sequence, but tracks signal changes rather than states.

State Graph for a C-element



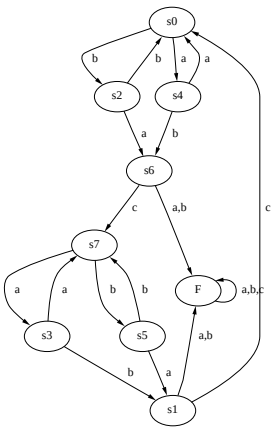
Traces Structures

- Set of all possible traces is represented using a *trace structure*.
- To verify hazard-freedom, use *prefix-closed trace structures*.
- Described using a four-tuple $\langle I, O, S, F \rangle$:
  - I is the set of input signals.
  - O is the set of output signals.
  - S is all traces which are considered successful.
  - F is all traces which are considered a failure.
- $A = I \cup O$ and $P = S \cup F$.

Receptive

- A trace structure must be *receptive*.
- It is receptive when the state of a circuit cannot prevent an input from happening (i.e., $PI \subseteq P$).

Receptive State Graph for a C-element



Inverse Delete

- Before composition of circuits must make their signal sets match.
- $T_1 = \langle I_1, O_1, S_1, F_1 \rangle$ and $T_2 = \langle I_2, O_2, S_2, F_2 \rangle$.
- If N is signals in A_2 and not in A_1, then add N to I_1 and extend S_1 and F_1 to allow events on signals in N at any time.
- Must also extend T_2 with those signals in A_1 but not in A_2.
- This is done by *inverse delete* function, denoted $del(N)^{-1}(x)$ where N is a set of signals and x is a set of traces.
- Function inserts elements of N^* between consecutive signals in x.
- This function can be extended to a trace structure as follows:

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

Composition

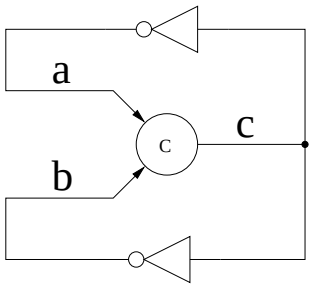
- Given two trace structures with *consistent signal sets* (i.e., $A_1 = A_2$ and $O_1 \cap O_2 = \emptyset$):

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

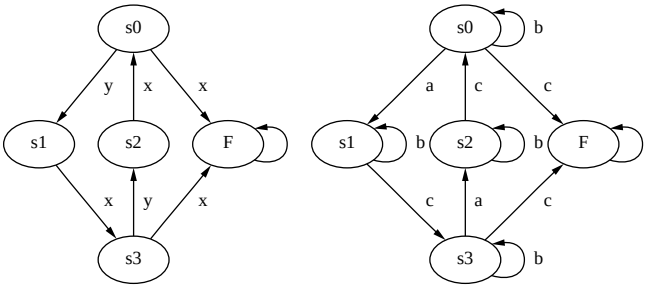
- Trace is success in composite when a success in both circuits.
- Trace is a failure when it is a failure in either circuit.
- Set of possible traces may be reduced ($P_1 \cap P_2$).
- Composition is defined as follows:

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

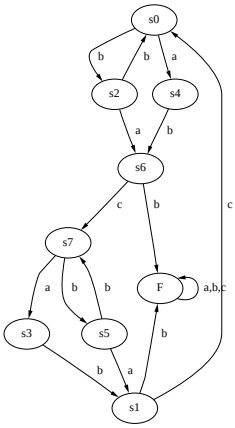
Composition Example



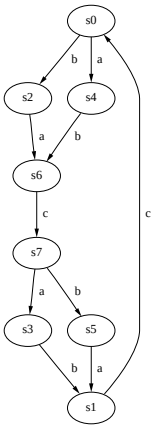
Inverter After Renaming and Inverse Deletion



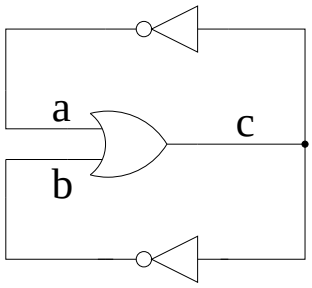
Composition of One Inverter and C-element



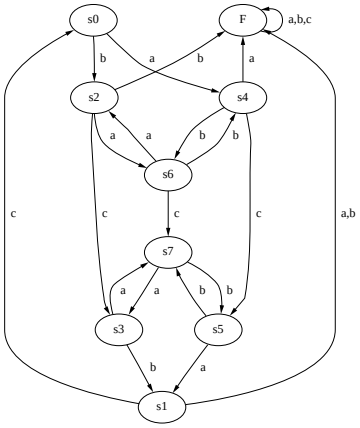
Complete Circuit



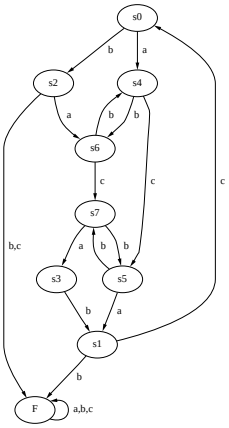
Composition Example2



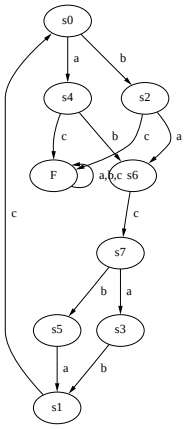
Receptive SG for an OR Gate



SG After Composing One Inverter with OR Gate



SG After Composing Both Inverters with OR Gate



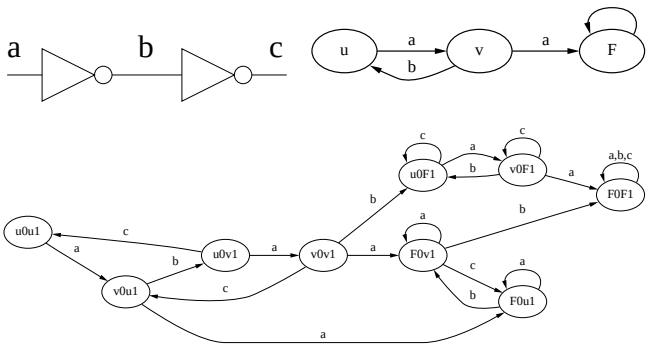
Conformance

- To verify that a circuit correctly implements a specification, we must show that T_I *conforms to* T_S (denoted $T_I \preceq T_S$).
- Must show that in any *environment*, T_E, where the specification is failure-free, the circuit is also failure-free.
- T_E is any trace structure with complementary inputs and outputs (i.e., $I_E = O_I = O_S$ and $O_E = I_I = I_S$).
- To check conformance, must show that for every possible T_E that if $T_E \cap T_S$ is failure-free then so is $T_E \cap T_I$.

Conformation Equivalence

- 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 $T_2 \preceq T_1$.
- If $T_1 \sim_C T_2$, it does not imply that $T_1 = T_2$.
- To make this true, use canonical prefix-closed trace structures.

Autofailures



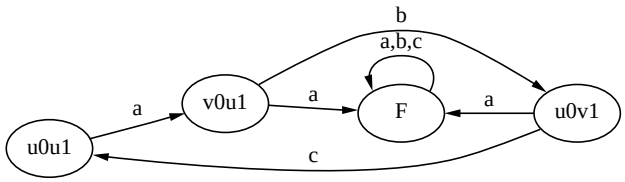
Autofailure Manifestation

- An *autofailure* is a trace x which if extended by a signal $y \in O$ then $xy \in F$.
- Also denoted $F/O \subseteq F$ where F/O is defined to be $\{x \mid \exists y \in O \,.\, xy \in F\}$.
- If $S \neq \emptyset$ then any failure trace has a prefix that is a success, and an input causes it to become a failure.
- If the environment sends a signal change which the circuit is not prepared for, we say that the circuit *chokes*.
- We must also add to the failure set any trace that has a failure as a prefix (i.e., $FA \subseteq F$).

Failure Exclusion

- *Failure exclusion* makes the success and failure sets disjoint.
- When trace occurs in both, circuit may or may not fail.
- Remove from success set any trace which is also a failure ($S = S - F$).

Two Inverters after Simplification



Canonical Prefix-Closed Trace Structures

- In a *canonical prefix-closed trace structure*:
  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 failure (i.e., $S \cap F = \emptyset$).
- Failure set is not necessary (i.e., $T = \langle I, O, S \rangle$).
- Determine the failure set as follows:

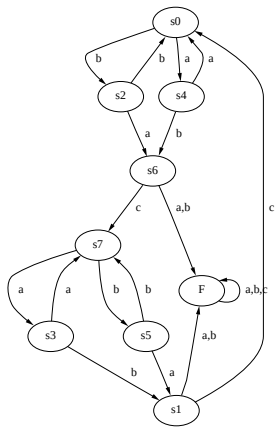
$$F = [(SI \cup \{\varepsilon\}) - S]A^*$$

- Any successful trace when extended with an input signal transition and is no longer found in the success set is a failure.
- Any such failure trace can be extended indefinitely and will always be a failure.

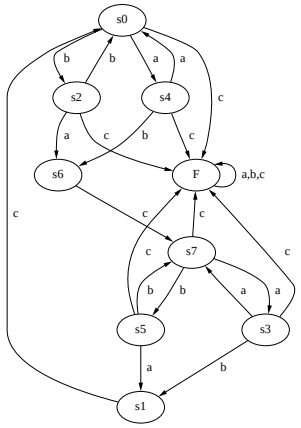
Mirrors

- To check $T_I \preceq T_S$, must check that in all environments that T_S is failure-free that T_I is also failure-free.
- Construct a unique worst-case environment called a *mirror* of T (denoted T^M).
- Mirror can be constructed by simply swapping the inputs and outputs (i.e., $I^M = O$, $O^M = I$, and $S^M = S$).
- If $T_I || T_S^M$ is failure-free, then $T_I \preceq T_S$.

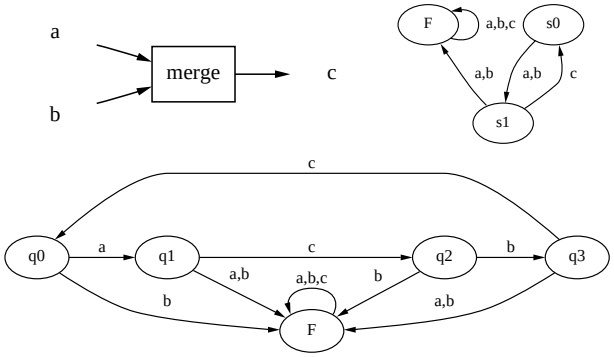
Receptive State Graph for a C-element



Mirror for a C-Element

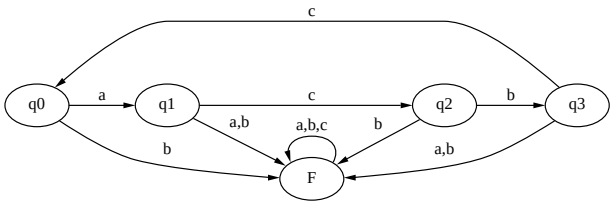


Example: Merge Element

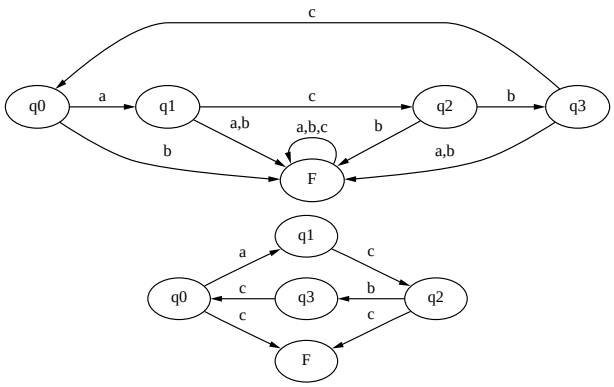


Can we replace alternating with general merge?

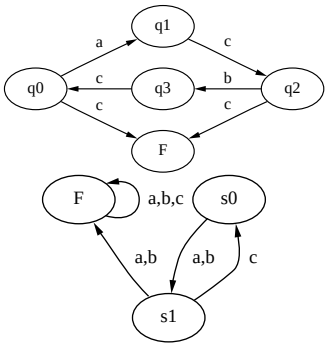
Can we replace alternating with general merge?



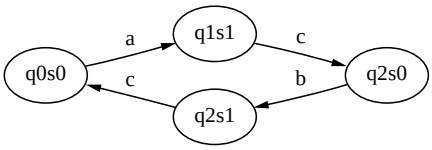
Can we replace alternating with general merge?



Can we replace alternating with general merge?

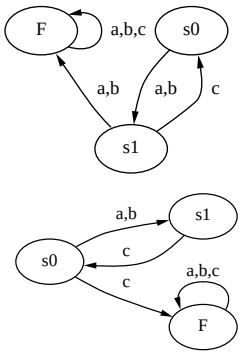


Can we replace alternating with general merge?

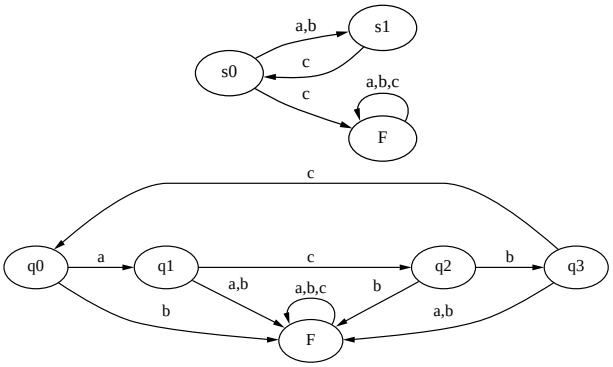


Can we replace general with alternating merge?

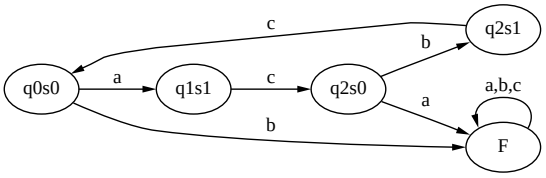
Can we replace general with alternating merge?



Can we replace general with alternating merge?



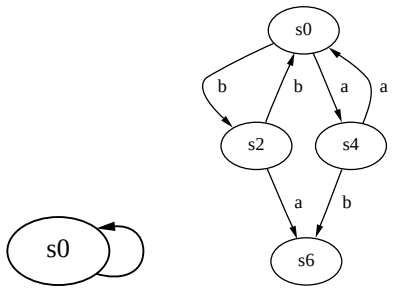
Can we replace general with alternating merge?



Limitations

- Only checks safety properties.
- If a circuit verifies, it means it does nothing bad.
- It does not mean, however, it does anything good.
- A “block of wood” accepts any input, but it never produces any output (i.e., $T = \langle I, O, I^* \rangle$).
- Assuming inputs and outputs are made to match, a block of wood would comform to any specification.

Block of Wood Example



Strong Conformance

- *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 $S_1 \supseteq S_2$.
- All successful traces of T_2 must be successful traces of T_1.

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:
  - $e_i \in A$, the set of signals.
  - $\tau_i \in \mathbf{Q}$, 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 $\tau_i > \tau$.

Advance Time

- 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 $\tau'' \geq \tau$.
- This means that after trace x, 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).

Safety Failures

- In timed case, must check that output is produced at an acceptable time.
- Consider $M = \langle I, O, S \rangle$ composed of $\{M_1, \ldots, M_n\}$, where $M_k = \langle I_k, O_k, S_k \rangle$.
- Consider $x = (x_1, \ldots, x_m)$, where $x_m = (w, \tau)$ and $w \in O_k$ for some $k \leq n$.
- x causes a failure if *advance_time*$(M, (x_1, \ldots, x_{m-1}), \tau)$, $x \in S_k$, but $x \notin S$.
- 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*.

Timing Failures

- A *timing failure* occurs when some module does not receive an input in time.
- Either some input fails to occur or occurs later than required.
- There are several ways to characterize timing failures formally, with each choice having different effects on the difficulty of verification.
- For the most general definition, it is no longer possible to use mirrors without some extra complexity.

Summary

- Protocol verification:
  - Linear temporal logic (LTL)
  - Timed LTL
- Circuit verification:
  - Trace structures
  - Conformance checking
  - Timed trace theory