Asynchronous Circuit Design

Chris J. Myers
Lecture 8: Verification
Chapter 8

Protocol Verification

- Specification for circuit usually tries 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 permissible 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:
  - Any signal u is a LTL formula.
  - If f and g are LTL formulas, so are:
    - \( \neg f \) (not)
    - \( f \land g \) (and)
    - \( f \implies g \) (next state operator)
    - \( f \mathcal{U} g \) (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.
- \( f \land g \) 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.
- \( f \mathcal{U} g \) 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

\[
\begin{align*}
  s_i \models u & \iff \lambda(s_i)(u) = 1 \\
  s_i \models \neg f & \iff s_i \models f \\
  s_i \models f \land g & \iff s_i \models f \text{ and } s_i \models g \\
  s_i \models \bigcirc f & \iff \text{ for all states } s_j \text{ such that } (s_i, s_j) \in \delta \text{, } s_j \models f \\
  s_i \models f \mathcal{U} g & \iff \text{ for all allowed sequences } (s_i, s_{i+1}, \ldots), \\
  & \exists j \geq i \land s_j \models g \land (\forall k \leq i < j \Rightarrow s_k \models f)
\end{align*}
\]
LTL Abbreviations

- $\Diamond f$ means $f$ will eventually become true in all allowed sequences starting in the current state.
  
  $$\Diamond f \equiv \text{true} \cup f$$

- $\Box f$ means $f$ is always true in all allowed sequences.
  
  $$\Box f \equiv \neg (\neg f)$$

- $f W g$ means $f$ is always true or until $g$.
  
  $$f W g \equiv (f \cup g) \cup \Box f$$

Desired Properties for a Passive/Active Wine Shop

- Should not raise $\text{ack\_wine}$ until $\text{req\_wine}$ goes high:
  
  $$\Box (\neg \text{ack\_wine} \Rightarrow (\neg \text{ack\_wine} \cup \text{req\_wine}))$$

- Once $\text{ack\_wine}$ is high, it must stay high until $\text{req\_wine}$ goes low:
  
  $$\Box (\text{ack\_wine} \Rightarrow (\text{ack\_wine} \cup \neg \text{req\_wine}))$$

- Once the shop has set $\text{req\_patron}$ high, it must hold it high until $\text{ack\_patron}$ goes high:
  
  $$\Box (\neg \text{req\_patron} \Rightarrow (\neg \text{req\_patron} \cup \text{ack\_patron}))$$

- Once the shop sets $\text{req\_patron}$ low, it must hold it low until $\text{ack\_patron}$ goes low:
  
  $$\Box (\neg \text{req\_patron} \Rightarrow (\neg \text{req\_patron} \cup \neg \text{ack\_patron}))$$

- The request and acknowledge wires on either side go high, they must be reset again:
  
  $$\Box (\text{req\_wine} \land \text{ack\_wine}) \Rightarrow (\Diamond (\neg \text{req\_wine} \land \neg \text{ack\_wine}))$$
  
  $$\Box (\neg \text{req\_patron} \land \text{ack\_patron}) \Rightarrow (\Diamond (\neg \text{req\_patron} \land \neg \text{ack\_patron}))$$

- The wine should not stay on the shelf forever, so after each bottle arrives, the patron should be called.
  
  $$\Box (\neg \text{ack\_wine} \Rightarrow \Diamond \neg \text{req\_patron})$$

- The patron should not arrive expecting wine in the shop before the wine has actually arrived.
  
  $$\Box (\neg \text{ack\_patron} \Rightarrow (\neg \text{ack\_patron} \cup \text{ack\_wine}))$$

- Should not raise $\text{ack\_wine}$ until $\text{req\_wine}$ goes high:
  
  $$\Box (\neg \text{ack\_wine} \Rightarrow (\neg \text{ack\_wine} \cup \text{req\_wine}))$$

- Once $\text{ack\_wine}$ is high, it must stay high until $\text{req\_wine}$ goes low:
  
  $$\Box (\text{ack\_wine} \Rightarrow (\text{ack\_wine} \cup \neg \text{req\_wine}))$$

- Once the shop has set $\text{req\_patron}$ high, it must hold it high until $\text{ack\_patron}$ goes high:
  
  $$\Box (\neg \text{req\_patron} \Rightarrow (\neg \text{req\_patron} \cup \text{ack\_patron}))$$

- Once the shop sets $\text{req\_patron}$ low, it must hold it low until $\text{ack\_patron}$ goes low:
  
  $$\Box (\neg \text{req\_patron} \Rightarrow (\neg \text{req\_patron} \cup \neg \text{ack\_patron}))$$
\( \neg \text{req\_patron} \Rightarrow (\neg \text{req\_patron} \cup \neg \text{ack\_patron}) \)

\( (\text{req\_wine} \land \text{ack\_wine}) \Rightarrow (\neg \text{req\_wine} \land \neg \text{ack\_wine}) \)

\( \text{ack\_wine} \Rightarrow \Diamond \text{req\_patron} \)

\( \neg \text{ack\_patron} \Rightarrow (\neg \text{ack\_patron} \cup \text{ack\_wine}) \)

\( \neg \text{ack\_patron} \Rightarrow (\neg \text{ack\_patron} \cup \text{ack\_wine}) \)
\[ \square (\neg \text{ack\_patron} \Rightarrow (\neg \text{ack\_patron} \bigcup \text{ack\_wine})) \]

Timed LTL

- If \( f \) 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.
- \( \Diamond \leq c f \) states that \( f \) becomes true in less than \( c \) time units.

Timed LTL Formulas

- Timed LTL formulas can be described recursively as follows:
  - Any atomic proposition \( p \in AP \) is a CTL formula.
  - If \( f \) and \( g \) are CTL formulas then so are:
    \begin{enumerate}
    \item \( \neg f \) (not)
    \item \( f \land g \) (and)
    \item \( f \bigcup \sim c g \)
    \end{enumerate}
  - Where \( \sim c \) is \begin{align*}
    \leq, \leq, =, \geq, > \end{align*}.
- There is no next time operator, since when time is dense, there can be no unique next time.

Timed LTL Abbreviations

- \( \Diamond = a f \equiv \Diamond \leq a f \)
- \( \Diamond < b f \equiv \Diamond \leq (b - a) f \)

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

Some Bounded Response Time Properties

- Once the request and acknowledge wires on either side go high, they must be reset again within 10 minutes:
  \[ \square ((\neg \text{req\_wine} \land \text{ack\_wine}) \Rightarrow \Diamond \leq 10 (\neg \text{req\_wine} \land \neg \text{ack\_wine})) \]
  \[ \square ((\text{req\_patron} \land \neg \text{ack\_patron}) \Rightarrow \Diamond \leq 10 (\neg \text{req\_patron} \land \text{ack\_patron})) \]
- 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:
  \[ \square (\text{ack\_wine} \Rightarrow \Diamond \leq 5 \text{ req\_patron}) \]

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.
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.

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 \( P = S \cup F \).

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

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 \( h_1 \) 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 \( \text{del}(N)^{-1}(x) \) 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:

\[
\text{del}(N)^{-1}(T) = \langle I \cup N, O, \text{del}(N)^{-1}(S), \text{del}(N)^{-1}(F) \rangle
\]
Composition

- Given two trace structures with consistent signal sets (i.e., \( A_1 = A_2 \) and \( O_1 \cap O_2 = \emptyset \)):
  \[
  T_1 \cap T_2 = \langle I_1 \cap I_2, O_1 \cup O_2, S_1 \cap S_2, (F_1 \cap F_2) \cup (F_2 \cap F_1) \rangle
  \]

- 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:
  \[
  T_1 || T_2 = del(A_2 - A_1)^{-1}(T_1) \cap del(A_1 - A_2)^{-1}(T_2)
  \]
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 trace $T_E$ that $T_E \cap T_S$ is failure-free then so is $T_E \cap T_I$.

Conformation Equivalence

Two trace structures $T_I$ and $T_S$ are conformation equivalent (denoted $T_I \sim C T_S$) when $T_I \preceq T_S$ and $T_S \preceq T_I$.

If $T_I \sim C T_S$, it does not imply that $T_I = T_S$.

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

![Two Inverters after Simplification](image)

Canonical Prefix-Closed Trace Structures

- In a *canonical prefix-closed trace structure*:
  - Autofailures are failures (i.e., $F/O \subseteq F$).
  - Once a trace fails, it remains a failure (i.e., $FA \subseteq F$).
  - No trace is both a success and failure (i.e., $S \cap F = \emptyset$).
  - Failure set is not necessary (i.e., $T = (I, O, S)$).
  - Determine the failure set as follows:
    \[ F = \left( \{ S \cup \{ \epsilon \} \} - S \right) A^* \]
  - 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

![Receptive State Graph for a C-element](image)
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 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., \( T = (I, O, O') \)).
- Assuming inputs and outputs are made to match, a block of wood would conform to any specification.
**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 $(e_i, \tau_i)$ such that:
  - $e_i \in E_i$ the set of events.
  - $\tau_i \in Q_i$ 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 $\tau \in Q$ there exists an index $i$ such that $\tau_i > \tau$.

**Safety Failures**

- In timed case, must check that output is produced at an acceptable time.
- Consider $M = (I, O, S)$ composed of $\{M_1, \ldots, M_n\}$, where $M_0 = (I_0, O_0, S_0)$.
- Consider $x = (x_1, \ldots, x_n)$, where $x_n = (w, \tau)$ and $w \in O_k$ for some $k \leq n$.
- $x$ causes a failure if $\text{advance_time}(M, (x_1, \ldots, x_{n-1}), \tau), x \in S_a, \text{but } x \not\in S$.
- 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**.

**Advance Time**

- Module $M$ allows time to advance to time $\tau$ if for each $w' \in I \cup O$ and $\tau' < \tau$ such that $x(w', \tau') \in S_1$, $x'(w', \tau') \in S_2$.
- We denote this by the predicate $\text{advance_time}(M, x, \tau)$.

**Strong Conformance**

- Strong conformance removes this problem.
- $T_1$ **conforms strongly to** $T_2$ (denoted $T_1 \preceq T_2$) if $T_1 \subseteq T_2$ and $S_1 \supseteq S_2$.
- All successful traces of $T_2$ must be successful traces of $T_1$.

**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