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.
- Simulation can again be used, but has the same problem.
- 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:
  1. Any signal u is a LTL formula.
  2. If f and g are LTL formulas, so are:
     (a) \( \neg f \) (not)
     (b) \( f \land g \) (and)
     (c) \( \Box f \) (next state operator)
     (d) \( f \mathbf{U} g \) (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.
- $f \land g$ 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.
- $f \lor g$ is true in a state $s_i$ when in all allowed sequences starting with $s_i$, $f$ is true until $g$ becomes true.

LTL Abbreviations

- $\bigcirc f$ means $f$ will eventually become true in all allowed sequences starting in the current state.
  $$\bigcirc f \equiv \text{true} \lor f$$
- $\Box f$ means $f$ is always true in all allowed sequences.
  $$\Box f \equiv \neg \bigcirc \neg f$$
- $f \bigcirc g$ means $f$ is always true or until $g$.
  $$f \bigcirc g \equiv (f \lor g) \land \Box f$$

Desired Properties for a Passive/Active Wine Shop

- Should not raise $\text{ack\_wine}$ until $\text{req\_wine}$ goes high:
  $$\Box(\neg \text{ack\_wine} \Rightarrow (\neg \text{ack\_wine} \lor \text{req\_wine}))$$
- Once $\text{ack\_wine}$ is high, it must stay high until $\text{req\_wine}$ goes low:
  $$\Box(\text{ack\_wine} \Rightarrow (\text{ack\_wine} \land \neg \text{req\_wine}))$$
- Once the shop has set $\text{req\_patron}$ high, it must hold it high until $\text{ack\_patron}$ goes high:
  $$\Box(\text{req\_patron} \Rightarrow (\neg \text{req\_patron} \lor \text{ack\_patron}))$$
- Once the shop sets $\text{req\_patron}$ low, it must hold it low until $\text{ack\_patron}$ goes low:
  $$\Box(\neg \text{req\_patron} \Rightarrow (\text{ack\_patron} \land \neg \text{req\_patron}))$$
Desired Properties for a Passive/Active Wine Shop

Once the request and acknowledgment wire are high, they must be reset again:

\[ \text{ack} \circ \text{wine} \in (\text{ack} \circ \text{wine} \cup \text{req} \circ \text{patron}) \]

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

\[ \text{ack} \circ \text{wine} \in (\text{ack} \circ \text{wine} \cup \text{req} \circ \text{patron}) \]

\[ \text{ack} \circ \text{patron} \in (\text{ack} \circ \text{patron} \cup \text{req} \circ \text{wine}) \]

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

\[ \text{ack} \circ \text{patron} \in (\text{ack} \circ \text{patron} \cup \text{req} \circ \text{wine}) \]

\[ \text{ack} \circ \text{patron} \in (\text{ack} \circ \text{patron} \cup \text{req} \circ \text{wine}) \]
Timed LTL

Timed LTL extends LTL by introducing time constraints. In timed LTL, each temporal operator is annotated with a time bound. To express bounded response time, it is necessary to extend the logic to include time bounds on events.

\( \text{Timed LTL} \)

\( (\neg \text{ack}-\text{pattern} \land \neg \text{ack-\text{wine}}) \Rightarrow (\neg \text{ack}-\text{pattern} \land \neg \text{ack-\text{wine}}) \)
Timed LTL Formulas

- Timed CTL 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:
     (a) $\neg f$ (not)
     (b) $f \land g$ (and)
     (c) $f U_{<c} g$
      where $\sim$ is $<, \leq, =, \geq, >$.
- There is no next time operator, since when time is dense, there can be no unique next time.

Some Bounded Response Time Properties

- Once the request and acknowledge wires on either side go high, they must be reset again within 10 minutes:
  $$\square((req_{wine} \land \neg ack_{wine}) \Rightarrow \Diamond_{\leq 10} (\neg req_{wine} \land \neg ack_{wine}))$$
  $$\square((req_{patron} \land \neg ack_{patron}) \Rightarrow \Diamond_{\leq 10} (\neg req_{patron} \land \neg ack_{patron}))$$
- 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:
  $$\square(ack_{wine} \Rightarrow \Diamond_{\leq 5} req_{patron})$$

Timed LTL Abbreviations

- Using the basic timed LTL primitives, we can also define temporal operators subscripted with time intervals.
  $$\Diamond_{\leq c} f \equiv \text{true} U_{\leq c} f$$
  $$\square_{\leq c} f \equiv \neg \Diamond_{\leq c} \neg f$$
  $$\Diamond_{(a,b]} f \equiv \Diamond_{a} \Diamond_{<(b-a)} f$$

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.

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 $(I, O, S, F)$:
  - $I$ 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

Composition
- Given two trace structures with consistent signal sets (i.e., $A_1 = A_2$ and $O_1 \cap O_2 = \emptyset$):
  $$T_1 \cap T_2 = (I_1 \cap I_2, O_1 \cup O_2, S_1 \cap S_2, (F_1 \cap P_2) \cup (F_2 \cap P_1))$$
- 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:
  $$T_1 || T_2 = \text{del}(A_2 - A_1)^{-1}(T_1) \cap \text{del}(A_1 - A_2)^{-1}(T_2)$$

Inverse Delete
- Before composition of circuits must make their signal sets match.
- $T_1 = (I_1, O_1, S_1, F_1)$ and $T_2 = (I_2, O_2, S_2, F_2)$.
- 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 $\text{del}(N)^{-1}(x)$ 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:
  $$\text{del}(N)^{-1}(T) = (I \cup N, O, \text{del}(N)^{-1}(S), \text{del}(N)^{-1}(F))$$

Composition Example
Inverter After Renaming and Inverse Deletion

Composition of One Inverter and C-element

Complete Circuit

Composition Example 2
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.

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 | \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., $FA \subseteq F$).

Autofailures

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

Canoncial 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., $T = (I, O, S)$).
- Determine the failure set as follows:
  $$F = [(SI \cup \{e\}) - S]A^*$$
- 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, then $T_I$ 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 $T_I || T^M_S$ is failure-free, then $T_I \preceq T_S$.

Mirror for a C-Element
Example: Merge Element

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., $T = \langle I, O, I' \rangle$).
- Assuming inputs and outputs are made to match, a block of wood would conform to any specification.
Slide 53

Block of Wood Example

Slide 54

Strong Conformance

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

Slide 55

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:
  - $e_i \in A$, the set of signals.
  - $\tau_i \in \mathbb{Q}$, 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 $\tau \in \mathbb{Q}$ there exists an index $i$ such that $\tau_i > \tau$.

Slide 56

Advance Time

- Module $M$ allows time to advance to time $\tau$ if for each $w' \in I \cup O$ and $\tau' < \tau$ such that $x(w', \tau') \in S$ 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 $\tau$ without needing an input or producing an output.
- We denote this by the predicate $\text{advance\_time}(M, x, \tau)$.
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 $\text{advance time}(M, (x_1, \ldots, x_{m-1}), \tau)$, $x \in S_k$, 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 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 CTL
- Circuit verification:
  - Trace structures
  - Conformance checking
  - Timed trace theory