8. Assertion-Based Design and Assertion Languages

Content

8.1 Assertion-Based Design
8.2 Introduction to ITL
8.3 Introduction to SVA
Appendix: Regular expressions
8.1 Assertion-Based Design

- Assertions, properties
- Software:
  - Mainly state-based
  - Only one time-point involved
- HDL assertions
  - VHDL
- Temporal Logic Assertions
  - May involve many (all) time-points
  - Safety/liveness properties
  - CTL (state-formulas)
  - LTL (path-formulas)
- Verification Languages
  - PSL (Property Specification Language)
  - SystemVerilog Assertions (SVA)
### 8.1 Assertion-based design

- **Assertions**

<table>
<thead>
<tr>
<th>Type</th>
<th>Time-points referenced</th>
<th>Examples</th>
<th>Proof procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>State-based</td>
<td>1</td>
<td>Software assertions</td>
<td>Simulation/Model-checking</td>
</tr>
<tr>
<td>Finite-window</td>
<td>1-20 (200)</td>
<td>Timing diagrams</td>
<td>Bounded model-checking</td>
</tr>
<tr>
<td>Infinite-window</td>
<td>all</td>
<td>Temporal logic formulas</td>
<td>Model-checking</td>
</tr>
</tbody>
</table>
Example of property-based verification (Winkelmann, Infineon)

- UMTS base-band station chip, 1024 parallel processes, configuration changes every 10 ms, Pipelining, 100 MHz, 2M gates, 70k lines of VHDL
- ~ 40k lines of assertions (properties) (560)
- Regression run 2 – 80 h
- 200 bugs found, 50 likely to escape simulation
8. Assertion-Based Design and Assertion Languages

8.1 Assertion-based design

- Assertion-(Property)-Based Design
- Assertions in VHDL
  - Assertions are checked during simulation
  - Violated assertions lead to messages
  - Syntax:
    ```
    assert CONDITION report "TEXT" severity LEVEL;
    ```
  - TEXT appears if CONDITION=false
  - Severity: note, warning, error, failure
    - Example:
      ```
      assert (S(1) and S(2)) /= '1' report "state 11"
      severity note;
      ```
      if state 11 is reached then the message "state 11" is reported
8. Assertion-Based Design and Assertion Languages

8.1 Assertion-based design

- 4 types of applications of assertions (2 soft, 2 hard)
  - **Cosimulation** after translation into HDL ("monitors")
    - e.g., FOCS tool (IBM) for complex assertions
  - **Formal verification**
    - e.g., (Bounded) Model-checker
8.1 Assertion-based design

- 4 types of applications of assertions (2 soft, 2 hard)
  - Generation of **hardware run-time monitors** and integration on-chip, e.g., FPGA

- **Synthesis** of hardware

![Diagram showing assertions, system description, circuits, and hardware synthesis processes]
8. Assertion-Based Design and Assertion Languages

8.1 Assertion-based design

- Monitors
  - Transform assertions into executable VHDL
  - Co-simulate assertions and monitor truth-value
    - Generate hardware-monitors + integrate on FPGA
    - Commercial solutions available
8. Assertion-Based Design and Assertion Languages

8.1 Assertion-based design

- **Hardware run-time monitors:**
  - Generate hardware monitors and integrate on, e.g., FPGA

\[ f_{1_t} \rightarrow \overline{f_{4_{t+2}}} \]
8.2 Introduction to ITL

- ITL: Interval Temporal Logic
  - Proprietary temporal language of the MV360 formal property checker of OneSpin Solutions
  - Specifically tailored to the needs of incomplete BMC
  - Provides only references in a finite time-window
Properties are specified as properties with an assume-part (the assumption) and a prove-part (the commitment)

```plaintext
property myproperty is
    assume: <assumptions>;
    prove:  <commitments>;
end property;
```
Each of the assumptions/commitments states a number of signal-values at some time-points by means of references to the time-variable $t$

- Example:

```plaintext
property myproperty is
assume:
    at t:  x = '0';
    at t+2: y = '1';
prove:
    at t+4: enable = '1';
    at t+7: ack = y;
end property;
```

The time-intervals of assumption and commitment may overlap!
8. Assertion-Based Design and Assertion Languages

8.2 Introduction to ITL

- VHDL or Verilog syntax is allowed for signals, constants, and functions
- `NEXT` and `PREV` are used to refer to next or previous time-points, respectively
  - The default values are 1
  - Example:

```vhdl
property myproperty is
  assume:
    at t: x = '0';
  prove:
    at t+2: NEXT(y) = PREV(x);
    at t+3: y = PREV(x,2);
end property;
```
during and within specify relationships that have to hold for all time-points or for at least one time-point in a reference time-interval, respectively

Example:

```plaintext
property myproperty is 
assume:
   at t: x = '0'; 
prove:
   during[t+1,t+4]:
      enable = PREV(y,2);
end property;
```
The `freeze` construct introduces a new alias for the value of a signal at a certain time-point; the alias can be referenced arbitrarily.

- **Example:**

```verbatim
property myproperty is
  freeze: y_t = y@t;
assume:
  at t: x = '0';
prove:
  within[t+1,t+4]: y = y_t;
end property;
```
8. Assertion-Based Design and Assertion Languages

8.2 Introduction to ITL

- ITL provides much more constructs like macros, index loops, etc.
- Semantics: "Implications between observed and expected behaviors at arbitrarily selectable time-points in a finite time-window"
8.3 Introduction to SVA

- **SVA**: SystemVerilog Assertions
  - Rich and expressive property language
  - Compatible with SystemVerilog
  - Part of SystemVerilog IEEE Standard 1800
  - "Similar" approaches:
    - OVL: Open Verification Library
    - Property Specification Language (PSL)
- Common idea: have a common and standardized language to express properties of a design
- **SVA resources:**
  - Foster/Krolnik/Lacey: Assertion-Based Design, Kluwer 2003 (also for OVL and PSL)
8. Assertion-Based Design and Assertion Languages

8.3 Introduction to SVA

- SVA is an assertion language based on SystemVerilog
  - Developed at Accellera as verification extension of SystemVerilog (IEEE1800-2005)
  - Used for formal verification as well as for the generation of simulation checkers (monitors)
Example 1:

\[ \text{req} \#\#1 \text{ ack} \#\#1 \sim \text{halt} \implies \text{grant} \#\#1 \text{ grant} \]

"After the sequence req=1; ack=1; halt=0, grant should be 1 for two time steps":

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>req</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ack</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>halt</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>grant</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

\[ \text{req} \#\#1 \text{ ack} \#\#1 \sim \text{halt} \quad \text{grant} \#\#1 \text{ grant} \]
Example 2:

The ack-signal must occur within 2 cycles after a req-signal (including reaction at the same step as req)

\[
\text{req} \rightarrow \#\#[^*0:2] \text{ ack}
\]
SVA is organized in four layers:

- **Boolean expression layer**
  - Just HDL-compatible Boolean expressions
  - and, or, etc.

- **Sequence layer**
  - Adds timing relationships

- **Property layer**
  - Definition of Property

- **Assertion directive layer**
  - Defines usage of property
The semantics of SVA is defined over legal execution paths (see Section 5.8) $\pi$ of a transition system $T$

- We use the notation $z_0, z_1, ...$ for a (legal) path $\pi$

- We follow the notational convention that $\pi^i$ denotes the $i$-th suffix $z_i, z_{i+1}, ...$ of a path

- In addition, $\pi^{i:j}$ with $i \leq j$ denotes the fragment from state $z_i$ to state $z_j$ of $\pi$

- $|\pi|$ denotes the length of a path

- The empty path of length 0 is formally denoted by $\varepsilon$
• **Boolean expressions**
  
  ↗ Some syntactic conventions:
  
  - `&`, `&&` and
  - `|`, `||` or
  - `~` negation
  - `=`, `==` equality, equivalence
  - `0`, `1` the Boolean constants

• **Semantics:**
  A Boolean expression `b` characterizes a path `π = z₀, z₁, ..., T,π |= b`, iff the state `z₀` is characterized by `b`
8. Assertion-Based Design and Assertion Languages

8.3 Introduction to SVA

• **Sequences**
  - Attempt to reflect the world of timing diagrams
  - Describe sequences of events
  - Not a property, but sort of a "filter"
  - Encapsulate behavior
  - Can be built from other sequences
  - Can be defined or used standalone

Syntax:
```
sequence NAME(args);
  <sequenceexpr>
endsequence;
```
• Sequences may be described using additional sequence operators
• Syntax definition ($r_1, r_2$ sequences; $b$ Boolean expression):
  ➤ Boolean expressions are sequences
  ➤ $r_1 \# 1 r_2$ (Sequence concatenation)
  ➤ $r_1 \# 0 r_2$ (Sequence fusion)
  ➤ $r_1[*0:$] (Consecutive repetition)
  ➤ $r_1$ or $r_2$ (Sequence or)
  ➤ $r_1$ and $r_2$ (Sequence and)
  ➤ $r_1$ within $r_2$ (Sequence within)
  ➤ $b$ throughout $r_2$ (Expression during sequence)
• The semantics is defined over finite paths $\pi$ (see the Reference Manual for the following)
Sequence concatenation: "r₁ ##₁ r₂"

Example: a, b Boolean expressions:
  a ##₁ b means that b holds at t and a holds at t-1

Semantics:
  \( T,\pi \models \{ r₁ ##₁ r₂ \} \) iff there exist paths \( \pi₁ \) and \( \pi₂ \) such that \( \pi = \pi₁\pi₂ \), \( T,\pi₁ \models r₁ \) and \( T,\pi₂ \models r₂ \)
More examples:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Sequence fusion: "$r_1 \#\#0 \ r_2$"

- $r_1$ and $r_2$ overlap by one cycle
  - Example: $(a \ #\#1 \ b) \ #\#0 \ (c \ #\#1 \ d)$ holds if $a$ holds in the first cycle, $b$ and $c$ in the second, $d$ in the third

Semantics:

$T, \pi |\!\!\!\!\!| = \{ r_1 \ #\#0 \ r_2 \}$ iff there exists paths $\pi_1$ and $\pi_2$ and state $z$ such that $\pi = \pi_1 z \pi_2$, $T, \pi_1 z |\!\!\!\!\!| = r_1$ and $T, z \pi_2 |\!\!\!\!\!| = r_2$
- **Sequence operators** *and* and *or*
  
  Semantics:
  \[ T, \pi |\Rightarrow r_1 \text{ and } r_2 \text{ iff } T, \pi |\Rightarrow r_1 \text{ and } T, \pi |\Rightarrow r_2 \]

  \( r_1 \text{ and } r_2 \text{ are not required to be of the same length } \)

  \[ T, \pi |\Rightarrow r_1 \text{ or } r_2 \text{ iff } T, \pi |\Rightarrow r_1 \text{ or } T, \pi |\Rightarrow r_2 \]
8. Assertion-Based Design and Assertion Languages

8.3 Introduction to SVA

- **Consecutive repetition operator** "s [*0 : $]"
  - Means zero or more repetitions of s
  - "s" arbitrary sequence
  - **Semantics:**
    \[ T, \pi |= s [*] \text{ iff either } \pi = \varepsilon \text{ or there exists paths } \pi_1, \pi_2, \ldots, \pi_j \text{ such that } \pi = \pi_1\pi_2\ldots\pi_j \text{ and for all } i, 1 \leq i \leq j, T, \pi_i |= s \]

- The "*" operator is originally the Kleene-operator for ordinary regular expressions; "a*" means "any sequence of a's including the empty sequence"; regular expressions are sequences of literals that can be constructed from concatenation and the *-operator
  - see Appendix of this Chapter
Sequence declaration

- Declared sequences have names and optional formal parameters
  - Example:
    ```
    sequence b_after_a (a, b);
     a ##[*0:2] b;
    endsequence;
    b_after_a(req, ack) |-> x is identical to
    req ##[*0:2] ack |-> x
    ```
8.3 Introduction to SVA

- **Property Layer**
  - Two types of properties: concurrent and immediate

- **Immediate Assertion:**
  - Direct test, whether expression true or false
  - Execution during simulation of behavioral code
  - Can only be used in procedures and functions
  - Limited instruction set

```
If (a>b) then
    assert(x) $display("PASS");
else
    $display("FAIL");
```
8.3 Introduction to SVA

- **Property Layer**
  - Two types of properties: concurrent and immediate

- **Concurrent Assertion:**
  - (Usually clocked) property check
  - Complete instruction set available
  - Can be used only at distinct places in the code
  - Can be used outside of procedures

Property definition, linked to SV-Module
8. Assertion-Based Design and Assertion Languages

8.3 Introduction to SVA

- **Property Layer**
  - Can be defined parametrically:

  Syntax: \[\text{property} \ \text{NAME}(\text{ARGS});\]

  \[
  \begin{align*}
  &<\text{property\_definition}> \\
  &\text{endproperty};
  \end{align*}
  \]

  - Basic operation is implication:
    - \(A \rightarrow B\) A implies B, B starts in the last cycle of A
    - \(A \Rightarrow B\) A implies B, B starts 1 cycle after the last cycle of A
8. Assertion-Based Design and Assertion Languages

8.3 Introduction to SVA

- **Suffix implication operator** "\( r_1 \rightarrow r_2 \)"
  - \( r_2 \) begins in the cycle when \( r_1 \) ends (overlapping by one cycle)
    - Example: \( \text{req} \#\#1 \ \text{ack} \#\#1 \sim \text{halt} \rightarrow \text{grant} \#\#1 \ \text{grant} \)

- **Semantics:**
  \( T, \pi \models r_1 \rightarrow r_2 \) iff for all \( j, 0 \leq j < |\pi| \), such that \( T, \pi^{0,j} \models r_1 \) there exists \( k, j \leq k \leq |\pi| \), and \( T, \pi^{j,k} \models r_2 \)

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 \\
\text{req} & 1 & - & - & - & - \\
\text{ack} & - & 1 & - & - & - \\
\text{halt} & - & - & 0 & - & - \\
\text{grant} & - & 1 & 1 & 1 & - \\
\end{array}
\]

\( T, \pi^{0,2} \models \text{req} \#\#1 \ \text{ack} \#\#1 \sim \text{halt} \)

\( T, \pi^{2,3} \models \text{grant} \#\#1 \ \text{grant} \)
- **Suffix next implication operator** "s1 |=> s2"
  
  ❯ Same as the suffix implication operator, but s2 begins in the cycle after the left sequence ends
  
  ❯ \( r_1 |=> r_2 = r_1 |-> 1 \#\#1 r_2 \)
  
  ❯ Example:
  
  \[
  \text{req} \#\#1 \text{ack} \#\#1 \sim \text{halt} |=> \text{grant}[*2]
  \]

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>req</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ack</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>halt</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>grant</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Semantics: Property satisfaction:

- Holds strongly:
  - no bad states have been seen
  - all future obligations have been met
  - the property will hold on any extension of the path

- Holds (but not strongly):
  - no bad states have been seen
  - all future obligations have been met
  - the property may or may not hold on any extension of the path

- Pending (Holds weakly):
  - no bad states have been seen
  - future obligations have not been met
  - (the property may or may not hold on any extension of the path)

- Fails:
  - a bad state has been seen
  - (future obligations may or may not have been met)
  - (the property may or may not hold on any extension of the path)
8. Assertion-Based Design and Assertion Languages

8.3 Introduction to SVA

- **Clocked** sequences and properties
- SVA provides constructs to restrict the evaluation of paths to reference signals (clocks) "@clk"

\[ (@\text{clk})(a ~\#1 ~b) \text{ holds for } t=3,6 \]
\[ (@\text{clk})(a ~\#1 ~b ~\#1 c) \text{ holds for } t=6 \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
The `posedge` and `negedge` functions can be used to specify rising and falling edges of clock-signals, respectively.

`(posedge clk)`
Properties define temporal relationships among Boolean expressions, sequences or properties itself.

Temporal property operators:
- The past-operator "$p_{past}(p)"
  - Takes us backward one clock cycle
- The always-operator "always p"
  - States that p always holds ($\sim$ LTL G)
  - Implicit operator for concurrent properties
- Property implication operators "p1 $\implies$ p2" and "p1 $\Rightarrow$ p2"
- And many more ...

The arguments may be either sequences or properties.
Property declaration

- Declared properties have names and optional formal parameters

  - Example:
    ```
    property mutex (boolean clk, a, b);
      always @(posedge clk) ~(a & b);
    endproperty;
    ```
Reset signals

Properties only useful if special conditions excluded.

"disable iff(c)" interrupts evaluation of property if c becomes true

Example:

```systeVerilog
property mutex (boolean clk, a, b, c);
    disable iff (c)
    always @(posedge clk) ~(a & b);
endproperty;
```
• **Assertion Layer**

Four codewords characterize a property:

- Codeword **ASSERT**: Property to be proven
- Codeword **ASSUMPTION**: Property is assumed
- Codeword **COVER**: Property is coverage-trigger
- Codeword **EXPECT**: Like **ASSERT**, but blocks execution

• **Examples:**

  P1: assert property (@posedge CLK) disable iff (COND) EXPR;
  P2: cover property Prop6(sig1,sig2,sig3);
  P3: assert (EXPR);
8.3 Introduction to SVA

- Summary
- SVA:
  - Rich and (quite) complex language
  - Idea: support "assertion-based design"
  - Automatic proof-tools for the full language do not exist currently (2010)
  - Automatic simulation checker generation for a large SVA-subset available
  - Sets of properties can not be simulated (in contrast to specifications in executable languages like SpecC, SystemC, etc.)
• Modern assertion-languages include regular expressions for the specification of properties
• A brief review of the relationship between automata and regular expressions …
• A Mealy-machine:
We consider the observable (legal, possible) sequences of input and output values (traces):

- "00" means i=0 and o=0, etc.
- An example trace is 00.00.11.10.10.10.00.11 …
- Formally, we consider the values 00, … as the "letters" of the alphabet \( A = \{00,10,11\} \)
- A "word" is formed by concatenating "." letters of the alphabet
- A language is a set of words formed according to some rules.
Not all words (e.g., 00.01) are observable in the example machine.

Regular expressions can be used to characterize the observable words of a machine.

For instance, the regular expression \([00*.11.10*.00]^*\) denotes all observable traces of the Mealy automaton.
• **Constructs of regular expressions**
  
  - Concatenation
  
  - *Kleene’s repetition operator (zero or arbitrary number of times)*
  
  - Alternative
    
    - Example:
      
      ```
      letter = {a, ..., z}, cipher = {0, ..., 9}, symbol = {-, _},
      ampersand = {@}, dot = {.},
      name = {letter}.{letter | cipher | symbol}* 
      simple e-mail address = 
      
      name.ampersand.name.{dot.name}*
      eveking@rs.tu-darmstadt.de
      ```

• The regular expression distinguishes between the observable and non-observable traces of a system
  
  - It characterizes, thus, the complete behaviour of the automaton
Based on a regular expression, we can also construct a device (monitor) which checks the correct behavior of the Mealy automaton.

The device is called a Deterministic Finite Acceptor (DFA).
A Deterministic Finite Acceptor (DFA):
- Has a starting state
- Has no outputs!
- Has two disjoint classes of (externally visible) states: accepting (green) and non-accepting (pink) states
- Receives letters of some alphabet \( A \) as input (e.g., \( A = \{00, 10, 11\} \) in our example)
- "Accepts" a finite word \( W \) over \( A \) iff the DFA is started with the first letter of \( W \) and the DFA is in an accepting state at the last letter of \( W \)
- For instance, the word 00.00.11.10.10.10.00.11 is accepted by the DFA above
- The word 00.01 is not accepted
- For every regular expression, a DFA can be constructed so that the DFA is in an accepting state iff the word is characterized by the regular expression
The DFA associated with a Mealy-machine is particularly easy to construct:

- All states of the Mealy-machine have corresponding accepting states in the DFA.
- There is a unique non-accepting state.
- All "wrong" (illegal, impossible) behaviours lead to this "sink" state.