IP Core Design

Lecture 11
Introduction to PSL

Juinn-Dar Huang, Ph.D.
Assistant Professor
jdhuang@mail.nctu.edu.tw

November 2004
Content

- Overview
- Fundamentals
- Boolean Layer
- Temporal Layer – SERE
- Temporal Layer – FL Property
- Modeling Layer
- Verification Layer
- Additional Information
Overview
Property Specification Language (PSL)

- Accellera adopted IBM’s Sugar as PSL standard in April 2002
- Sugar 2.0 is a formalism to reason about behavior over time
- Uses of Sugar
  - documentation
    - easy to read
    - precise semantics
  - input to simulation tools
  - input to formal verification tools
    - an extension of CTL and LTL
Language Goal

- Easy to learn, read and write
- Mathematically precise
  - rigorously well defined formal syntax and semantics
- Sufficiently expressive
  - permitting the specification of a large class of “real” world design properties
- Known efficient underlying algorithms
  - in simulation
  - in model checking (with reasonable complexity)
    - FL can be compiled down to LTL
    - OBE can be compiled down to CTL
Layer Structure of Sugar

- **Boolean layer**
  - reason about states of the design

- **Temporal layer**
  - reason about behavior of the design over time

- **Modeling layer**
  - model behavior of design inputs
    - rose( ), fell( ), prev( ), next( )
  - model auxiliary state variables and state machines

- **Verification layer**
  - provide directives to the verification tool
    - assert, assume, cover, …
  - group Sugar constructs into verification unit
Flavors

- Sugar comes in three flavors
  - Verilog
  - VHDL
  - EDL (Environment Description Language)
- The flavor determines the syntax of Boolean and Modeling layers
- Temporal and Verification layers are identical in all 3 flavors
  - actually, a slight difference does exist

This lecture tastes Verilog flavor
Fundamentals
Temporal Layer (1/2)

- **Boolean Expressions**
  - expressions evaluated over a single state

- **Sugar Extended Regular Expressions (SERE)**
  - expressions evaluated over a bounded sequence of states

- **Foundation Language**
  - expressions evaluated over finite or infinite sequence of states

- **Optional Branching Extension (OBE)**
  - expression evaluated over infinite trees of states
    (relevant for formal verification only)
Temporal Layer (2/2)

Boolean Expressions

Sugar Extended Regular Expressions (SERE)

Foundation Language

Optional Branching Extension (OBE)
Linear vs. Branching Semantics

• Linear semantics
  – Foundation Language (FL)
  – can be compiled down to pure LTL formulas
  – reasoning about computation paths
  – can be checked in both simulation and formal verification

• Branching semantics
  – Optional Branching Extension (OBE)
  – can be compiled down to pure CTL formulas
  – reasoning about computation trees
  – can be checked in formal verification only

This lecture focuses on FL only
Hold and Hold Tightly

• Hold
  – a Boolean expression/sequence/property holds in the first cycle of a path iff the path exhibits the behavior described by the Boolean expression/sequence/property

• Hold Tightly
  – a sequence holds tightly along a finite path iff the path exhibits the behavior described by the sequence
Clocked vs. Unclocked Evaluation

- Both are supported in Sugar
- Equivalent to
  - cycle-based vs. event-driven
  - synchronous vs. asynchronous evaluation
- Always clocked evaluation for static formal model checkers
Safety vs. Liveness Property

• Safety property
  – specifies an invariant over the limited states
  – whenever \texttt{req} is asserted, \texttt{ack} is asserted within 3 cycles

• Liveness property
  – specifies an eventuality that is unbound in time
  – whenever \texttt{req} is asserted, \texttt{ack} is asserted sometime in the future
Strong vs. Weak Operator

- Some operators have a terminating condition that comes at an unknown time
  - busy shall be asserted until done is asserted
  - is done eventually asserted?
- Strong operator requires it eventually occurs
  - (busy until! done)
- Week operator puts no requirement
  - (busy until done)
Simple Subset

- The subset of FL can be easily checked in simulation
  - key: monotonic advancement of time (causality)
  - always (a \rightarrow \text{next}[3] b) \quad O
  - always ((a \& \text{next}[3] b) \rightarrow c) \quad X
  - negation (!) is applied only to Booleans
  - never and eventually! are applied only to Booleans or SEREs
  - LHS of logical and/or/implication (\rightarrow) is Boolean
  - both sides of logical iff (\leftrightarrow) are Booleans
  - RHS of non-overlapping until* is Boolean
  - both sides of overlapping until_* are Booleans
  - both sides of before*/before_* are Booleans
Boolean Layer
Boolean Expression

Boolean ::= 
    HDL_Expression 
    | PSL_Expression

• Value of the HDL_Expression
  – as the condition of an if statement
PSL Expression

• Boolean layer also includes certain PSL expressions of Boolean type
  – predefined functions, `rose()`, `fell()`, `prev()`, `next()` in Modeling Layer
  – *endpoint* variable
Clocked Expressions

- Boolean expression can be clocked
  – provide synchronous check mechanism

```plaintext
assert always (req -> next ack) @ (posedge clk);
cover {req; ack; !req; !ack} @ (posedge clk);
```

- Default clock

```plaintext
default clock = Boolean (event type)

default clock = (posedge clk);
assert always (req -> next ack);
cover {req; ack; !req; !ack};
```
Temporal Layer - SERE
SERE

SERE : Sugar Extended Regular Expression

SERE ::= Boolean | Sequence

Sequence ::= { SERE }

• A Sequence is a SERE
• A SERE is not a Sequence
Clocked SERE

Left Blank Intentionally
SERE Concatenation (;)

SERE ::= SERE ; SERE

• A ; B holds tightly on a path iff
  – there is a future cycle n (n ≥ 1)
  – A holds tightly form the 1st cycle to the n-th cycle and
  – B holds tightly on the path starting from the (n+1)-th cycle

{a ; b}      {a[*3] ; b}  
{a ; b[*3]}  {a ; {b ; c}}
{{a ; b} ; c} {a ; b ; c}
More About SERE (1/3)

\[ \{ \text{req}; \text{busy}; \text{gnt} \} \]

This diagram is described by the SERE
More About SERE (2/3)

\{ req; busy; gnt \}

This diagram is also described by the SERE
More About SERE (3/3)

\[
\{ \text{req} \& \text{!busy} \& \text{!gnt}; \\
\text{!req} \& \text{busy} \& \text{!gnt}; \\
\text{!req} \& \text{!busy} \& \text{gnt} \}
\]

If we want to describe only this diagram we have to change the SERE.
SERE Repetition Operators

- The repetition operators ([ ]) are used to describe repeated concatenation of the same SERE.
- Three kind of repetitions:
  - consecutive repetition ([* ])
  - non-consecutive repetition ([= ])
  - (non-consecutive) goto repetition ([-> ])

Provide more succinct syntax

Support in LDV5.0
Example

\( \{ req; busy; busy; busy; busy; busy; gnt \} \)

\( \{ req; busy[^4]; gnt \} \)
SERE Consecutive Repetition ([* ])

SERE ::= 
    SERE[*[Count]]
    |   SERE[+]
    |   [+][Count]
    |   [+]  

Count ::= 
    Non-negative integer  
    |   Range  

Range ::= LowBound : HighBound
LowBound ::= Non-Negative integer | 0
HighBound ::= Non-Negative integer | inf

infinity
## Semantics

### Semantics of consecutive repetition ($0 \leq n \leq m$)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A[*n]$</td>
<td>$A$ repeats $n$ times</td>
</tr>
<tr>
<td>$A[*n:m]$</td>
<td>$A$ repeats between $n$ to $m$ times</td>
</tr>
<tr>
<td>$A[*:m] = A[*0:m]$</td>
<td>$A$ repeats at most $m$ times (including 0 time)</td>
</tr>
<tr>
<td>$A[*n:] = A[*n:inf]$</td>
<td>$A$ repeats at least $n$ times</td>
</tr>
<tr>
<td>$A[<em>] = A[</em>:] = A[*0:inf]$</td>
<td>$A$ repeats any number of times (including 0)</td>
</tr>
<tr>
<td>$A[+] = A[*1:]$</td>
<td>$A$ repeats at least 1 time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[*n]$</td>
<td>Any sequence of $n$ cycles</td>
</tr>
<tr>
<td>$[*n:m]$</td>
<td>Any sequence of $n$ to $m$ cycles</td>
</tr>
<tr>
<td>$[*:m] = [*0:m]$</td>
<td>Any sequence of at most $m$ cycles (including 0)</td>
</tr>
<tr>
<td>$[*n:] = [*n:inf]$</td>
<td>Any sequence of at least $n$ cycles</td>
</tr>
<tr>
<td>$[<em>] = [</em>:] = [*0:inf]$</td>
<td>Any sequence (including empty one)</td>
</tr>
<tr>
<td>$[+] = [*1:]$</td>
<td>Any sequence of at least 1 cycle</td>
</tr>
</tbody>
</table>
SERE Non-Consecutive Repetition ([= ])

SERE ::=  
  Boolean [=Count] (Note: Boolean, not SERE)

Count ::=  
  Non-negative integer
  | Range

Range ::= LowBound : HighBound

LowBound ::= Non-Negative integer | 0

HighBound ::= Non-Negative integer | inf
Semantics of non-consecutive repetition ($0 \leq n \leq m$)

- $A[=n]$  
  A occurs $n$ times

- $A[=n:m]$  
  A occurs between $n$ to $m$ times

- $A[:m] = A[=0:m]$  
  A occurs at most $m$ times (including 0 time)

  A occurs at least $n$ times

- $A[=:] = A[=0:inf]$  
  A occurs any number of times (including 0)

$A[=]$ is not allowed
SERE Goto Repetition ([-> ])

SERE ::= Boolean [->] [Count]  (Note: Boolean, not SERE)

Count ::= Positive integer  (Note: Positive, not Non-negative)
        | Positive Range

Range ::= LowBound : HighBound

LowBound ::= Positive integer | 1  (Note: not 0)

HighBound ::= Positive integer | inf
Semantics

Semantics of goto repetition ($1 \leq n \leq m$)

- **A[->n]**: A occurs $n$ times
- **A[->n:m]**: A occurs between $n$ to $m$ times
- **A[->1:m] = A[->:m]**: A occurs at most $m$ times (excluding 0 time)
- **A[->n:inf] = A[->n:]**: A occurs at least $n$ times
- **A[->1:inf] = A[->:]**: A occurs one or more times
- **A[->] = A[->1]**: A occurs exactly one time
Comparisons between \([=] \) and \([-\rightarrow]\)

\[
\{a[=3]\} \Rightarrow \{b\} \quad \text{holds for Case 1 & 2}
\]
\[
\{a[-\rightarrow3]\} \Rightarrow \{b\} \quad \text{holds for Case 2 only}
\]

\([=] \) does not require the last occurrence is at the last cycle

\([-\rightarrow] \) requires the last occurrence is at the last cycle
Sequence Fusion (:) 

\[
\text{SERE} ::= \text{Sequence : Sequence} \\
\text{Sequence} ::= \{ \text{SERE} \}
\]

- Two sequences overlap by one cycle
- The 2nd sequence starts at the cycle in which the 1st sequence completes

\[
\{\{a;b\} ; \{c;d\}\} \quad \{\{a;b\} : \{c;d\}\}
\]
Sequence Or ($)}

\[ \text{SERE} ::= \]
\[ \text{Sequence} \mid \text{Sequence} \]
\[ \text{Sequence} ::= \{ \text{SERE} \} \]

- For sequence \( A \) and \( B \), \( A \mid B \) holds tightly on a path iff at least one of \( A \) or \( B \) holds tightly on the path

\[ \{ \{a[*3]\} \mid \{b[*3]\} \} \]

Support in LDV5.0

\[ \text{fails} \]
Sequence Non-Length-Matching And (&)

SERE ::= Sequence & Sequence
Sequence ::= { SERE }

- For sequence A and B, A & B holds tightly on a path iff A holds tightly on the path and B holds tightly on a prefix of the path; or vice versa

\{\{a[*]\} & \{b[*]\}\}
**Sequence Length-Matching And (&&)**

SERE ::= 
Sequence && Sequence
Sequence ::= \{ SERE \}

- For sequence A and B, A && B holds tightly on a path iff both A and B hold tightly on the path

\{{a[*]} && {b[*]}\}

---

Support in LDV5.0
Named Sequence

Sequence_Declaration ::=  
    sequence Name [(Formal_Parameter_List)] = Sequence ;
Formal_Parameter_List ::=  
    Formal_Parameter { ; Formal_Parameter}
Formal_Parameter ::=  
    Parameter_Kind Name { , Name}
Parameter_Kind ::=  
    const | boolean | sequence

LDV5.0 does not support parameterized named sequence yet
Example of Named Sequence

Declaration:

Bus Arbitration
sequence BusArb(boolean br, bg; const n) =
{br; (br && !bg)[*0:n]; br && bg};

Read Operation
sequence ReadCycle(sequence ba; boolean bb, ar, dr) =
{ba; {bb[*]} && {ar[->]; dr[->]}; !bb};

Instantiation:

BusArb(breq, back, 3)
ReadCycle(BusArb(breq, back, 3), breq, ardy, drdy)
Endpoint

Endpoint_Declaration ::= 
  endpoint Name [(Formal_Parameter_List)] = Sequence ;

Formal_Parameter_List ::= 
  Formal_Parameter {; Formal_Parameter}

Formal_Parameter ::= 
  Parameter_Kind Name {, Name}

Parameter_Kind ::= 
  const | boolean | sequence

- An endpoint is a **Boolean**-valued variable that indicates when the given sequence completes
  - the value is **true** only at the **last** cycle of the sequence

LDV5.0 does not support parameterized endpoint yet
Example of Endpoint

Declaration:

```plaintext
endpoint ActiveLowReset(boolean rst_n, clk; const n) =
    {(rst_n == 1'b0)[*n:]; rst_n == 1'b1} @ (posedge clk);
```

Instantiation:

```plaintext
ActiveLowReset(rst_n, mclk, 3);
```
Temporal Layer – FL Property
FL Property

FL_Property ::= Boolean | (FL_Property)
Clocked FL Property

FL_Property ::= 
    FL_Property @ Clocked_Boolean
| FL_Property @ Clocked_Boolean!

• Provide cycle-based synchronous check capability
• Strong form requires at least one clock tick
• Nested clocked Boolean is allowed
  - ((a until b) @ (posedge clk1)) @ (posedge clk2)

Support in LDV5.0
always

\[ FL_{Property} ::= \\
   \text{always } FL_{Property} \\
   \mid \text{always } \text{Sequence} \]

specifies that the property/sequence \textit{always} holds \textit{at all times}

always a;

always \{a; b; c\};

\[ \begin{array}{c}
   a \\
   b \\
   c \\
   \vdots
\end{array} \]
**never**

\[
\text{FL\_Property ::= never FL\_Property}
\]

\[
\text{never Sequence}
\]

specifies that the property/sequence **never** holds at all times

---

**Simple Subset**

the operand of **never** is restricted to a Boolean expression or a Sequence
eventually!

\[
\text{FL\_Property} ::= \begin{align*}
\text{eventually! FL\_Property} \\
\mid \text{eventually! Sequence}
\end{align*}
\]

specifies that the property/sequence holds at the current cycle or at some future cycle.

**Simple Subset**

the operand of `eventually!` is restricted to a Boolean expression or a Sequence.
next (1/2)

\[
\text{FL\_Property ::= next! FL\_Property} \\
| \text{next FL\_Property} \\
| \text{next[Number](FL\_Property)} \\
| \text{next[Number](FL\_Property)}
\]

specifies that the property operand holds at some next cycle
next (2/2)

• next! holds
  – there is a next cycle and the property operand holds at the next cycle → Cond1

• next holds
  – there is no next cycle or Cond1

• next![n] holds
  – there is an n-th next cycle and the property operand holds at the n-th next cycle → Cond2

• next[n] holds
  – there is no n-th next cycle or Cond2

Note
• the number n can be 0; means the current cycle
• next(p) ≡ next[1](p)
• next!(p) ≡ next![1](p)
**next_a**

\[
\text{FL_Property} ::= \\
\quad \text{next_a!}[\text{Range}](\text{FL_Property}) \\
\quad | \quad \text{next_a}[\text{Range}](\text{FL_Property})
\]

\[
\text{Range} ::= i:j \quad (\text{Note: } 0 \leq i \leq j; i \text{ and } j \neq \text{inf})
\]

specifies that the property operand holds at all cycles between the i-th and the j-th next cycle, inclusively

- **next_a! [i:j]** is a strong operator
  - the j-th next cycle **shall exist**
- **next_a [i:j]** is a weak operator
  - the i-th through j-th next cycles **do not necessarily exist**
next_e

FL_Property ::= 
    next_e! [Range] (FL_Property) 
| next_e [Range] (FL_Property)

Range ::= i:j  \[Note: 0 \leq i \leq j; i and j \neq \text{inf}\]

specifies that the property operand holds at least once between the i-th and the j-th next cycle, inclusively

- next_e! [i:j] is a strong operator
  - the property operand shall hold at least once no matter there is a j-th next cycle or not
- next_e [i:j] is a weak operator
  - the property operand may not hold if there is no j-th next cycle

Not Support in LDV5.0
**next_event (1/2)**

\[
\text{FL\_Property} ::= \\
\quad \text{next\_event!}(\text{Boolean})(\text{FL\_Property}) \\
\quad | \quad \text{next\_event}(\text{Boolean})(\text{FL\_Property}) \\
\quad | \quad \text{next\_event!}(\text{Boolean})[\text{Number}](\text{FL\_Property}) \\
\quad | \quad \text{next\_event}(\text{Boolean})[\text{Number}](\text{FL\_Property}) \\
\text{Number} ::= \text{Positive integer}
\]

specifies that the property operand holds at some next occurrence of the Boolean operand

**Note**

- Number cannot be 0
- the occurrence is counting from the current cycle

Not Support in LDV5.0
next_event (2/2)

• next_event! holds
  – Boolean operand holds at least once and the property operand holds at the 1st occurrence of Boolean operand → Cond1
• next_event holds
  – Boolean operand never holds or Cond1
• next_event! [n] holds
  – Boolean operand holds at least n times and the property operand holds at the n-th occurrence of the Boolean operand → Cond2
• next_event[n] holds
  – there is no n-th occurrence of the Boolean operand or Cond2

Note
• next_event(b)(p) ≡ next_event(b)[1](p)
• next_event(true)(p) ≡ next[0](p)
next_event_a

FL_Property ::= 
    next_event_a!(Boolean)[Range](FL_Property) 
    | next_event_a(Boolean)[Range](FL_Property)

Range ::= i:j  (Note: 1 ≤ i ≤ j; i and j ≠ inf)

specifies that the property operand holds at all occurrences between the i-th and the j-th occurrence, inclusively

• next_event_a![i:j] is a strong operator
  – the j-th next occurrence shall exist
• next_event_a[i:j] is a weak operator
  – the i-th through j-th next occurrences do not necessarily exist

Not Support in LDV5.0
**next_event_e**

\[ \text{FL	extunderscore Property} ::= \]
\[ \text{next	extunderscore e}[\text{Range}](\text{FL	extunderscore Property}) \]
\[ | \text{next	extunderscore e}[\text{Range}](\text{FL	extunderscore Property}) \]

\[ \text{Range} ::= i:j \]  
(\text{Note: } 0 \leq i \leq j; i \text{ and } j \neq \text{inf})

specifies that the property operand holds at least one occurrence between the i-th and the j-th occurrence, inclusively

- \text{next	extunderscore e}[i:j] is a strong operator
  - the property operand shall hold at least once no matter there is a j-th occurrence or not
- \text{next	extunderscore e}[i:j] is a weak operator
  - the property operand may not hold if there is no j-th occurrence

Not Support in LDV5.0
abort

\[ \text{FL\_Property ::= FL\_Property abort Boolean} \]

- An abort property holds at the given cycle iff
  - the property operand holds at the given cycle; or
  - the series of cycles does not contradict the property
    - starting from the given current cycle, and
    - ending with the cycle in which the Boolean operand holds

always ((req -> eventually! ack) abort cancel);
until (1/2)

\[
FL_{\text{Property}} ::= \\
\quad FL_{\text{Property}} \ until! \enspace FL_{\text{Property}} \\
\mid FL_{\text{Property}} \ until \enspace FL_{\text{Property}} \\
\mid FL_{\text{Property}} \ until!\_\_ \enspace FL_{\text{Property}} \\
\mid FL_{\text{Property}} \ until\_\_ \enspace FL_{\text{Property}}
\]

specifies that the left property holds \textit{until} the right property holds

**Simple Subset**

right operand of \textit{until!} and \textit{until} : Boolean expression
both operands of \textit{until!\_} and \textit{until\_} : Boolean expression

Support in LDV5.0
until (2/2)

- **until!** and **until!_** are strong operators
  - the right property operand shall eventually hold
- **until** and **until_** are weak operators
  - the right property operand can never hold
  assume a is always true, then
  
  always (a until! b); => fails if b is never true
  always (a until b); => holds if b is never true

- **until!** and **until** are **non-inclusive** operators
  - the left property operand holds up to, but not necessarily including, the cycle in which the right one holds

- **until!_** and **until_** are **inclusive** operators
  - the left property operand holds up to and including, the cycle in which the right one holds

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a until b holds
a until_ b fails
before (1/2)

\[
\text{FL\_Property ::=}
\]

\[
\begin{align*}
& \text{FL\_Property before! FL\_Property} \\
& | \text{FL\_Property before FL\_Property} \\
& | \text{FL\_Property before!_ FL\_Property} \\
& | \text{FL\_Property before_ FL\_Property}
\end{align*}
\]

specifies that the left property holds \texttt{before} the right property holds

Simple Subset
both operands of \texttt{before*} is restricted to a Boolean expression

Not Support in LDV5.0
before (2/2)

- `before!` and `before!_` are strong operators
  - the left property operand shall eventually hold
- `before` and `before_` are weak operators
  - the left property operand can never hold

  Assume `b` is always false, then
  
  - `always (a before! b);` => fails if `a` is never true
  - `always (a before b);` => holds if `a` is never true

- `before!` and `before` are non-inclusive operators
  - the left property operand holds strictly before the right one holds
- `before!_` and `before_` are inclusive operators
  - the left property operand holds before or at the same cycle as the right one holds

```
  a | b
  |   a before b fails
  | a before_ b holds
```
Suffix Implication (1/5)

FL_Property ::= 
  Sequence(FL_Property) ●
  | Sequence  |-> Sequence! ●
  | Sequence  |-> Sequence ●
  | Sequence  |=> Sequence! ●
  | Sequence  |=> Sequence ●

specifies that the right property/sequence holds if the left sequence holds
Suffix Implication (2/5)

- sequence (FL_Property) holds in the given cycle
  - the sequence does not hold in the given cycle; or
  - the property operand holds in any cycle C such that the sequence holds tightly from the given cycle to C

- sequence |→ sequence! holds in the given cycle
  - the left sequence does not hold in the given cycle; or
  - in any cycle C such that the left sequence holds tightly from the given cycle to C, the right sequence holds

- sequence |→ sequence holds in the given cycle
  - the left sequence does not hold in the given cycle; or
  - in any cycle C such that the left sequence holds tightly from the given cycle to C,
    - the right sequence holds
    - any prefix of the path beginning from C can be extended so that the right sequence holds tightly on the extended path
Suffix Implication (3/5)

\{a, b\}(d) \text{ holds in cycle } n

\{a, b\} \implies \{c, d\} \text{ holds in cycle } n

\{a[=2]\} \implies \{c, d\} \text{ holds in cycle } n

\{a[=2]\} \implies \{c, d\} \text{ also holds in cycle } n
Suffix Implication (4/5)

- sequence $\Rightarrow\text{sequence}!$ holds in the given cycle
  - the left sequence does not hold in the given cycle; or
  - in any cycle $C$ such that the left sequence holds tightly from the given cycle to $C$
    - the right sequence holds in the cycle after $C$

- sequence $\Rightarrow\text{sequence}$ holds in the given cycle
  - the left sequence does not hold in the given cycle; or
  - in any cycle $C$ such that the left sequence holds tightly from the given cycle to $C$,
    - the right sequence holds in the cycle after $C$
    - any prefix of the path beginning from the cycle after $C$ can be extended so that the right sequence holds tightly on the extended path
Suffix Implication (5/5)

\{a, b\} \implies \{c, d\} \text{ holds in cycle } n

\{a[=2]\} \implies \{c, d\}

\text{also holds in cycle } n
**whilenot (1/3)**

\[ FL_{\text{Property}} ::= \]

- \( \text{whilenot!}(\text{Boolean}) \) Sequence
- \( \text{whilenot}(\text{Boolean}) \) Sequence
- \( \text{whilenot!\_}(\text{Boolean}) \) Sequence
- \( \text{whilenot\_}(\text{Boolean}) \) Sequence

specifies that the sequence **holds tightly on the path from the current cycle to the cycle that the Boolean holds**

Not Support in LDV5.0
**whilenot (2/3)**

- **whilenot!(Boolean) Sequence holds**
  - Boolean holds at the given cycle and Sequence holds on an empty path (Cond1); or
  - a cycle C that Boolean holds first time such that Sequence holds tightly from the given cycle to the cycle before C (Cond2)

- **whilenot(Boolean) Sequence holds**
  - Cond1 or Cond2 or
  - any prefix of that path starting from the given cycle can be extended such that Sequence holds tightly on the extended path
whilenot (3/3)

• **whilenot!_**(*(Boolean)*) **Sequence holds**
  – Boolean holds at the given cycle and **Sequence holds at the given cycle** *(Cond1)*; or
  – a cycle C that Boolean holds *first time* such that **Sequence holds tightly from the given cycle to C** *(Cond2)*

• **whilenot_**(*Boolean*) **Sequence holds**
  – **Cond1** or **Cond2** or
  – any prefix of that path starting from the given cycle can be extended such that **Sequence holds tightly on the extended path**
within (1/3)

\[
FL\_Property ::= \\
\quad \text{within!}(\text{Seq_or_Bool, Boolean}) \text{ Sequence} \\
\quad | \quad \text{within}(\text{Seq_or_Bool, Boolean}) \text{ Sequence} \\
\quad | \quad \text{within!}_!(\text{Seq_or_Bool, Boolean}) \text{ Sequence} \\
\quad | \quad \text{within}_!(\text{Seq_or_Bool, Boolean}) \text{ Sequence} \\
\text{Seq_or_Bool ::=} \\
\quad \text{Sequence} \\
\quad | \quad \text{Boolean}
\]

specifies that Sequence holds tightly on the path from the cycle C, such that Seq_or_Bool holds tightly from the current cycle to C, to the cycle that Boolean holds.
within (2/3)

- within! (S_or_B, Boolean) Sequence holds
  - S_or_B does not hold at the given cycle (Cond1); or
  - in any cycle C such that S_or_B holds tightly from the given cycle to C, either
    - Boolean holds at C and Sequence holds on an empty path (Cond2)
    - there is a cycle D subsequent to cycle C such that Boolean holds first time and Sequence holds tightly from C to the cycle before D (Cond3)
- within (S_or_B, Boolean) Sequence holds
  - Cond1 or
  - in any cycle C such that S_or_B holds tightly from the given cycle to C, either
    - cond2 or cond3 or
    - any prefix of that path starting from C can be extended such that Sequence holds tightly on the extended path
within (3/3)

- within!_$(S$\_or\_B, Boolean) Sequence holds
  - $S$\_or\_B does not hold at the given cycle (Cond1); or
  - in any cycle $C$ such that $S$\_or\_B holds tightly from the given cycle to $C$, either
    - Boolean holds at $C$ and Sequence holds in $C$ (Cond2)
    - there is a cycle $D$ subsequent to cycle $C$ such that Boolean holds first time and Sequence holds tightly from $C$ to $D$ (Cond3)

- within_$(S$\_or\_B, Boolean) Sequence holds
  - Cond1 or
  - in any cycle $C$ such that $S$\_or\_B holds tightly from the given cycle to $C$, either
    - cond2 or cond3 or
    - any prefix of that path starting from $C$ can be extended such that Sequence holds tightly on the extended path
Logical Implication

\[
\text{FL\_Property ::= FL\_Property \rightarrow FL\_Property}
\]

specifies that the right property shall hold if the left property holds

- A logical implication property holds at the given cycle iff
  - left property operand does not hold at the given cycle; or
  - right property operand holds at the given cycle

Simple Subset

the left operand of logical implication is restricted to a Boolean expression
**Logical iff**

\[
\text{FL\_Property} ::= \quad \text{FL\_Property} \leftrightarrow \text{FL\_Property}
\]

specifies that either properties holds, or neither properties holds

- A logical iff property holds at the given cycle iff
  - both property operands hold at the given cycle; or
  - neither property operands hold at the given cycle

**Simple Subset**

both operands of *logical iff*

is restricted to a Boolean expression

Not Support in LDV5.0
Logical and

\[
\text{FL\_Property ::= FL\_Property && FL\_Property}
\]

specifies that both properties shall hold

Simple Subset

the left operands of \text{logical and} is restricted to a Boolean expression

Not Support in LDV5.0
Logical or

```
FL_Property ::= FL_Property || FL_Property
```

specifies that at least one of properties shall hold

**Simple Subset**

the left operands of **logical or** is restricted to a Boolean expression

Not Support in LDV5.0
Logical not

\[
\text{FL\_Property ::= ! FL\_Property}
\]

specifies that the specified property shall not hold

Simple Subset
the operand of logical not
is restricted to a Boolean expression
## LTL Operators

Standard LTL operators can also be used in PSL

<table>
<thead>
<tr>
<th>Standard LTL Operator</th>
<th>Equivalent PSL Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>next</td>
</tr>
<tr>
<td>X!</td>
<td>next!</td>
</tr>
<tr>
<td>F</td>
<td>eventually!</td>
</tr>
<tr>
<td>G</td>
<td>always</td>
</tr>
<tr>
<td>U</td>
<td>until!</td>
</tr>
<tr>
<td>W</td>
<td>until</td>
</tr>
</tbody>
</table>

Not Support in LDV5.0
OBE Properties

Left Blank Intentionally
Replicated Property (1/2)

Property ::= 
    Replicator Property

Replicator ::= 
    forall Name [[Range]] in ValueSet :

ValueSet ::= 
    {ValueRange {, ValueRange}}
    |  boolean

ValueRange ::= 
    Value
    |  Range

Range ::= 
    i:j  (Note: 0 ≤ i ≤ j; i and j ≠ inf)
Replicated Property (2/2)

forall i in boolean: f(i)
   f(true), f(false)

forall i in {j, k} : f(i)
   f(j), f(j+1), ..., f(k)

forall i[0:1] in {4, 5} : f(i)
   f({4, 4}), f({4, 5}), f({5, 4}), f({5, 5})

forall i[0:3] in boolean :
   (req && (din == i)) -> next(dout == i)

forall j in {0:7} :
  forall k in {0:3} :
      f(j, k)
Named Property

Property_Declaration ::= 
    property Name [(Formal_Parameter_List)] = Property ;
Formal_Parameter_List ::= 
    Formal_Parameter {; Formal_Parameter}
Formal_Parameter ::= 
    Parameter_Kind Name {, Name}
Parameter_Kind ::= 
    const | boolean | property | sequence

property ResultAfterN(boolean start, stop; property result; const n) = 
    always ((start -> next[n] (result)) @ (posedge clk) abort stop);

ResultAfterN(read_req, cancel, eventually! ack, 3)

LDV5.0 does not support parameterized property yet
## Operator Precedence

<table>
<thead>
<tr>
<th>Category</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDL (Verilog) operators</td>
<td>(in addition, property &amp; &amp;</td>
</tr>
<tr>
<td>Clocking operator</td>
<td>@</td>
</tr>
<tr>
<td>SERE construction operators</td>
<td>; [*] [=] [-&gt;]</td>
</tr>
<tr>
<td>Sequence composition operators</td>
<td>:</td>
</tr>
<tr>
<td>FL implication operators</td>
<td></td>
</tr>
<tr>
<td>FL occurrence operators</td>
<td>always never eventually! next! within* whilenot*</td>
</tr>
<tr>
<td>Termination operators</td>
<td>abort until* before*</td>
</tr>
</tbody>
</table>
Modeling Layer
Verilog Flavored Modeling Layer

- Verilog-Flavored modeling layer consists of a *synthesizable subset* defined by IEEE P1364.1
- This subset is also extended with
  - integer ranges
  - structures
  - non-determinism
  - built-in functions
    - `rose()`, `fell()`, `prev()`, and `next()`
Built-In Functions (1/2)

• rose(Boolean)
  – similar to posedge
  – \( \text{rose}(b) \equiv b \land \neg \text{prev}(b) \)

• fell(Boolean)
  – similar to negedge
  – \( \text{fell}(b) \equiv \neg b \land \text{prev}(b) \)
Built-In Functions (2/2)

- **prev(expression)**
  - return the value of the expression in the previous cycle
  - prev(b), the 1-bit previous value of b
  - prev(vec[31:0]), the 32-bit previous value of vec

- **prev(expression, n) (n > 0)**
  - return the value of the expression in the n-th previous cycle
  - prev(b, 3)

- **next(expression)**
  - non-causality
Other Extensions

Left Blank Intentionally
Verification Layer
Objective

- Provide **directives** informing the verification tools what to do with the specified properties
- Provide constructs **grouping** related PSL statements and directives
- Directives
  - assert
  - assume
  - assume_guarantee
  - restrict
  - restrict_guarantee
  - cover
  - fairness and strong fairness
assert and assume

Assert Statement ::= assert Property ;

Assume Statement ::= assume FL_Property ;

Assume Guarantee Statement ::= assume_guarantee FL_Property ;

\[\text{assume} \quad \text{always} \ (\text{ack} \rightarrow \text{next} \ !\text{ack})\]

\[\text{assert} \quad \text{always} \ (\text{ack} \rightarrow \text{next} \ !\text{ack})\]

\[\text{assume}_\text{guarantee} \quad \text{always} \ (\text{ack} \rightarrow \text{next} \ !\text{ack})\]

Support in LDV5.0
restrict

Restrict_Statement ::= restrict Sequence ;

Restrict_Guarantee_Statement ::= restrict_guarantee Sequence ;

• Used to initialize the design before checking assertions

restrict {!rst; rst[*3]; !rst[*]};
restrict_guarantee {!rst; rst[*3]; !rst[*]};
**cover**

Cover Statement ::= cover Sequence ;

- Used to **check** the a given sequence was covered during the verification process
- It is very useful for **functional coverage**

```plaintext
cover {beg_trans; !end_trans[*];
    beg_trans & end_trans};
// to check if a transaction starts at the same cycle in which the previous transaction completes
```
fairness

Left Blank Intentionally

Not Support in LDV5.0
Verification Unit

Verification_Unit ::= VUnitType Name [(Hierarchical_HDL_Name)] { {Inherit_Spec} {VUnit_Item} }
VUnitType ::= vunit | vprop | vmode
Hierarchical_HDL_Name ::= Module_Name | Hierarchical_Instance_Name
Inherit_Spec ::= inherit VUnit_Name {, VUnit_Name} ;
Vunit_Item ::= Modeling_Layer_Declaration_or_Statement | PSL_Declaration (Note: Property, endpoint, sequence, clock) | Verification_Directive

LDV5.0 does not support vprop/vmode and verification unit inheritance yet
vunit, vprop and vmode

• vprop unit
  – declare assertions
  – shall not contain a directive that is not an assert directive
  – shall not inherit from vunit or vmode

• vmode unit
  – declare constraints
  – shall not contain assert directives
  – shall not inherit from vunit or vmode

• vunit unit
  – can contain all directives without restrictions

• A default verification unit can exist
  – the name shall be default
  – the type shall be vmode
Verification Unit Binding

• Bound to an instance name
  \[\text{vunit ex1a(top.i1) } \{
    \text{assert never (ena & enb); } \}\]
is equivalent to
  \[\text{vunit ex1b } \{
    \text{assert never (top.i1.ena & top.i1.enb); } \}\]

• Bound to a module name  
  (tip: preferred one)
  if \text{mod1} is a module; \text{top.i1} and \text{top.i2} are two instances
  \[\text{vunit ex2a (mod1) } \{
    \text{assert never (ena & enb); } \}\]
is equivalent to
  \[\text{vunit ex2b } \{
    \text{assert never (top.i1.ena & top.i1.enb); }
    \text{assert never (top.i2.ena & top.i2.enb); } \}\]
Verification Unit Inheritance

Example

vunit ex1a(top.i1) {
    assert never (read_en & write_en); }
vunit ex1b(top.i1.i2) {
    inherit ex1a;
    assert never (ena & enb); }

is equivalent to

vunit ex1c {
    assert never (top.i1.read_en & top.i1.write_en);
    assert never (top.i1.i2.ena & top.i1.i2.enb); }
Verification Unit Scoping Rules

• To resolve name conflict in a verification unit
• Precedence (low to high)
  – in the design
  – in the default verification unit
  – in an inherited verification unit
  – in the current verification unit
Additional Information
Syntactic Sugaring

• Sugar has a **core** of operators which determine its expressive power

• Other operators are **syntactic sugaring** of the core operators
  – for example,
    
    (\texttt{f1 until\_ f2}) is translated to (\texttt{f1 until f1\&\&f2})
    
    (\texttt{f1 before\ f2}) is translated to (\texttt{!f2 until f1\&\&!f2})
    
    (\texttt{f1 before\_ f2}) is translated to (\texttt{!f2 until f1})

• A tool needs to implement
  – only the core operators
  – macro expansion of the syntactic sugaring operators
Reference Information

- Sugar Home
- Tutorial for Sugar
  - www.eedesign.com/story/OEG20020509S0075
- PSL LRM
  - http://www.eda.org/vfv/docs/psl_lrm-1.01.pdf
- Accellera
  - www.accellera.org
- Using PSL/Sugar with Verilog and VHDL
  - Ben Cohen, May 2003, ISBN: 0-9705394-4-4
- Cadence Incisive Simulator
  - Simulation-Based Assertion Checking Guide
  - Simulation-Based Assertion Checking Tutorial
  - Simulation-Based Assertion Writing Tutorial