System Verilog Assertions

SE767 – Vérification et Test

Ulrich Kühne

02/03/2020

Outline

A Practical Verification Language?

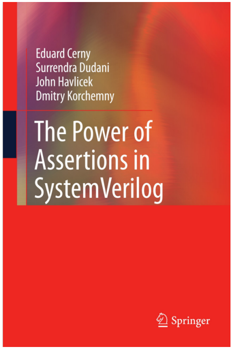
- LTL and CTL have emerged from theoretical interest
- Bound to specific complexity classes and equivalence notions
- Nested CTL/LTL properties are hard to understand
- Subtle semantic differences

$$\mathbf{F}\,\mathbf{X}\,p \equiv \mathbf{X}\,\mathbf{F}\,p \equiv \mathbf{AX}\,\mathbf{AF}\,p \not\equiv \mathbf{AF}\,\mathbf{AX}\,p$$

$$\mathbf{F}\,\mathbf{G}\,p \not\equiv \mathbf{AF}\,\mathbf{AG}\,p$$

System Verilog Assertions

- Industrial standard (IEEE 1800-2012)
- Embedded in SystemVerilog HDL
- Superset of LTL
- Sequences and regular expressions
- Supports simulation and formal verification



Basic Property Structure

```
// basic property structure
property foo;
  @(posedge clk) disable iff (rst)
    expr;
endproperty // foo

// verification directives
assert_foo: assert property(foo);
assume_foo: assume property(foo);
```

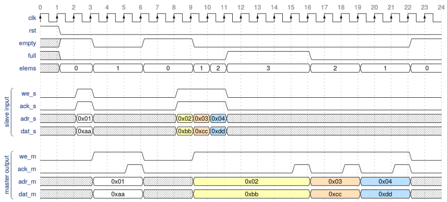
Past Values and Value Changes

- Value of a signal in the preceding cycle:
  `$past(a)`
- Shortcut for rising edge:
  `$rose(a)` is equal to `!$past(a) && a`
- Shortcut for falling edge:
  `$fell(a)` is equal to `$past(a) && !a`
- Shortcut for stable signal:
  `$stable(a)` is equal to `$past(a) == a`
- Shortcut for changed signal:
  `$changed(a)` is equal to `$past(a) ^ a`

Outline

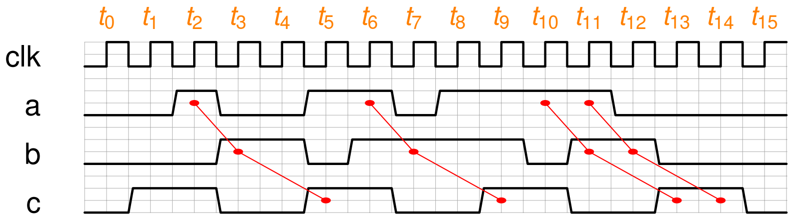
Sequentially Extended Regular Expressions (SERE)

- Typical use case: Chains of events
- Awkward to describe in LTL
- Intuitive description by regular expressions
- Syntax resembles known languages (bash, Python, . . .)



Sequences

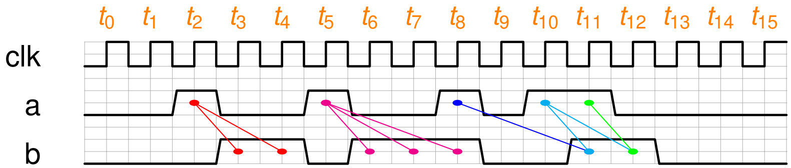
```
a ##1 b ##2 c
```



- Find all matching cycles...

Variable Delay

```
a ##[1:3] b
```



t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12} t_{13} t_{14} t_{15}
clk
a
b

- Find all matches…

Repetition

- Consider sequence:

```
a ##1 a ##1 a ##1 b ##1 b
```

- Shortcut for repeating sequence:

```
a[*3] ##1 b[*2]
```

- Variable repetition:

```
a[*1:3]
```

Assertion Semantics

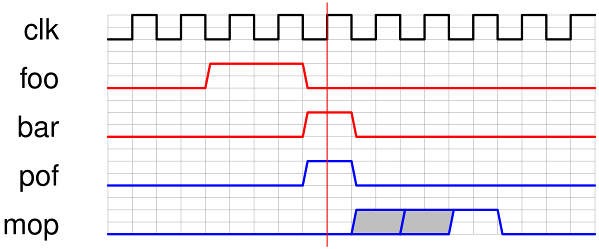
```
property foo;
   @(posedge clk)
     a ##[1:3] b;
endproperty

assert_foo: assert property(foo);
```

- What are we actually verifying here?
- Sequence `a ##[1:3] b` must match in all cycles
- Implicit always operator (**G** in LTL)

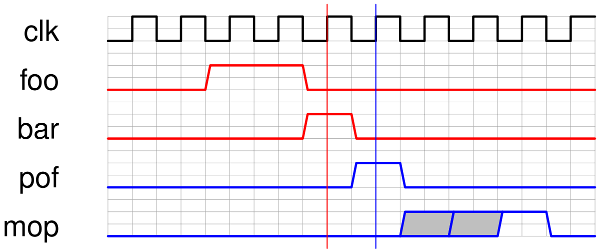
Suffix Implication

```
// suffix implication
foo ##1 bar |-> pof ##[1:3] mop
```



Non-Overlapping Suffix Implication

```
// non-overlapping suffix implication
foo ##1 bar |=> pof ##[1:3] mop
```



Example

Prop. 1: “Whenever signal `rdy` is asserted, it must stay asserted for 5 clock cycles”

```
property rdy_stable;
   @(posedge clk)
     !rdy ##1 rdy |=> rdy[*4];
endproperty
```

Outline

Infinity

- Special symbol `$` for infinity
- Can be used in variable delay and repetition
- `a[*]` is a shortcut for `a[*0:$]`
- `a[+]` is a shortcut for `a[*1:$]`

Exercise: What is the meaning of this sequence?

```
(start ##1 busy[*] ##1 done)[+]
```

Infinity

Prop. 2: “Whenever signal `busy` is asserted,
`rdy` must be asserted eventually.”

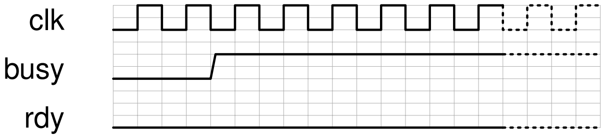
```
property rdy_after_busy;
  @(posedge clk)
    busy |-> ##[0:$] rdy;
endproperty
```

This property is wrong!

Infinity

- This assertion has no counter-example

```
busy |-> ##[0:$] rdy
```



Strength

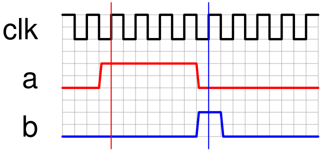
- Use of `strong(...)` operator
- Enforces a match before the end of evaluation (which is infinity in formal verification)
- Weak and strong versions of many operators

```
A1: assert property (busy |-> ##[0:$] rdy);
A2: assert property (busy |-> strong( ##[0:$] rdy ));
A3: assert property (busy |-> eventually rdy);
A4: assert property (busy |-> s_eventually rdy);
```

Until

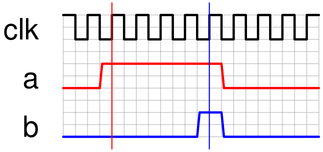
`a until b`

`a s_until b`

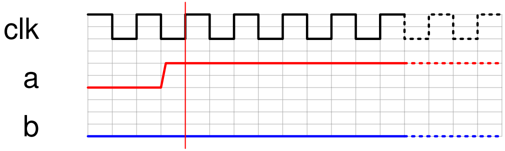


`a until_with b`

`a s_until_with b`



Attention: Weak until operators allow infinite wait!



Outline

Goto Repetition

Prop. 3: "After signal `write` is serviced by `ack`, signal `ready` should be asserted."

```
(write ##1 !ack[*] ##1 ack) |=> ready
```

Prop. 4: "After signal `wr_burst` is serviced twice by `ack`, signal `ready` should be asserted."

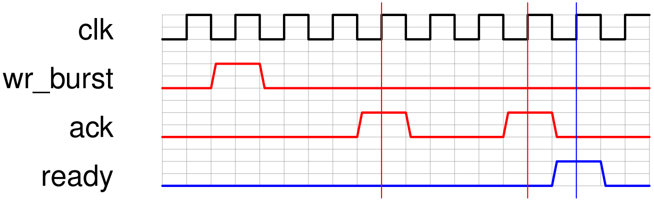
```
(wr_burst ##1 !ack[*] ##1 ack ##1 !ack[*] ##1 ack) |=> ready
```

```
(wr_burst ##1 (!ack[*] ##1 ack)[*2] ) |=> ready
```

```
wr_burst ##1 ack[->2] |=> ready
```

Goto Repetition

```
wr_burst ##1 ack[->2] |=> ready
```



Within / Throughout

Prop. 5: “Throughout the whole burst cycle, the signal `ready` should be low.”

```
!ready throughout (wr_burst ##1 ack[->2])
```

Prop. 6: “Within a granted bus cycle, a write transaction should be completed.”

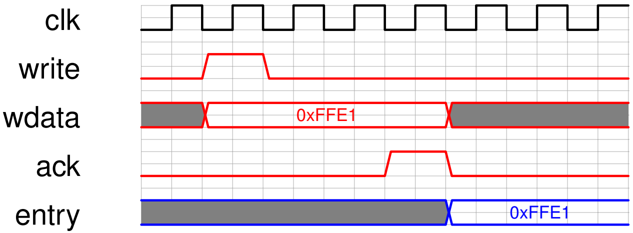
```
(write ##1 ack[->1]) within (gnt ##1 !gnt[->1])
```

This property is (probably) wrong!

```
(write ##1 ack[->1] ##1 1) within (gnt ##1 !gnt[->1])
```

Local Variables

Prop. 7: “After a completed write transaction, the value of `wdata` is stored in the register `entry`.”



Local Variables

Prop. 7: “After a completed write transaction, the value of `wdata` is stored in the register `entry`.”

```
property foo;
  logic[15:0] tmp;
  @(posedge clk)
    (write, tmp = wdata) ##1 ack[->1] |=>
      entry == tmp;
endproperty
```

Practical Exercise

- Formalization of a textual specification
- Verification and debugging with *qformal*
- See exercise on website
  `https://sen.enst.fr/verification-formelle`

References I

Cerny, E., Dudani, S., Havlicek, J., and Korchemny, D. (2015).
SVA: The Power of Assertions in SystemVerilog.
Springer.