



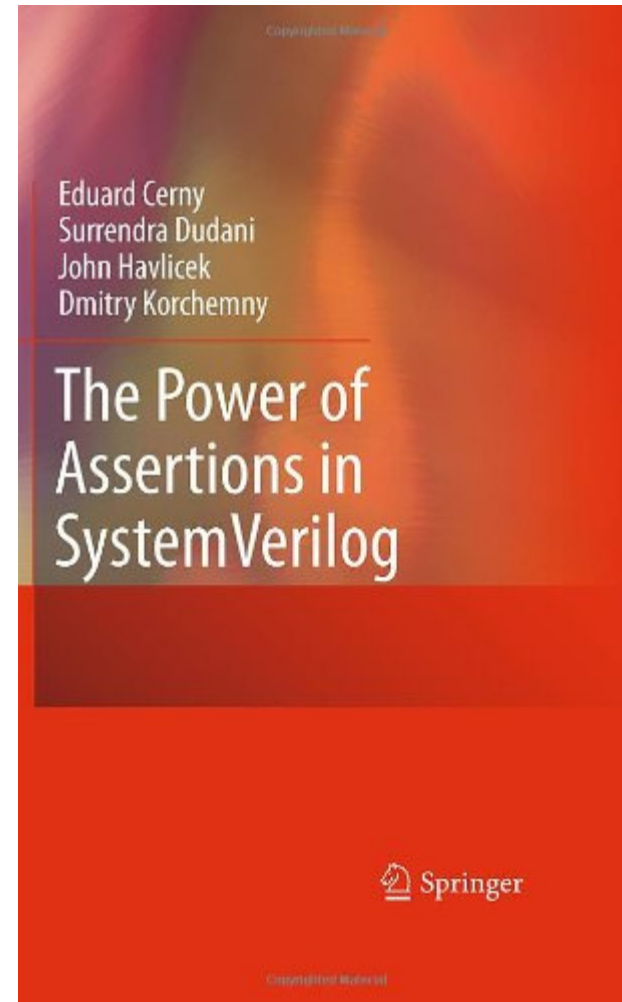
# SYSTEMVERILOG ASSERTIONS FOR FORMAL VERIFICATION

---

Dmitry Korchemny, Intel Corp.

**HVC2013**, November 4, 2013, Haifa

- Most of the examples used in this tutorial are borrowed from our SVA book



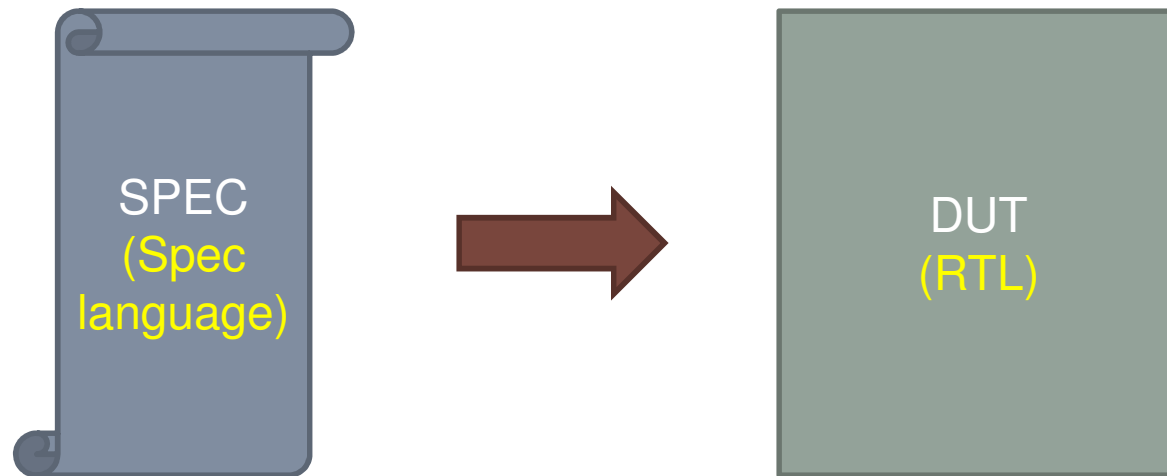
# Agenda

- Introduction
- Formal verification model. LTL properties
- Assertion statements
- Sequences and properties
- Clocks and resets
- Assertion system functions
- Metalanguage and checkers
- Local variables
- Recursive properties
- Efficiency and methodology tips
- Future directions and challenges

# INTRODUCTION

---

# Hardware Verification Task



- Does DUT meet the spec?
- Simulation
  - Does DUT meet the spec for **given** input stimuli?
- Formal Verification (FV)
  - Does DUT meet the spec for **any** legal input stimuli?

# SystemVerilog Assertions (SVA)

- SystemVerilog (proliferation of Verilog) is a unified hardware design, specification, and verification language
  - RTL/gate/transistor level
  - Assertions (SVA)
  - Testbench (SVTB)
  - API
- SVA is a formal specification language
  - Native part of SystemVerilog [SV12]
  - Good for simulation and formal verification

# SVA Standardization History

- 2003
  - Basic assertion features defined
- 2005
  - Improved assertion semantics
- 2009
  - Major changes in the language: deferred assertions, LTL support, checkers
- 2012
  - Improved checker usability, final assertions, enhancements in bit-vector system functions and in assertion control
- Part of SystemVerilog standardization (IEEE 1800)

# SVA vs. PSL

- Formal semantics of SVA is (almost) consistent with the formal semantics of PSL [PSL10]
- Meta-language layers are quite different (e.g., checkers vs. vunits)
- SVA has well-defined simulation semantics; tightly integrated with other parts of SystemVerilog



# FORMAL VERIFICATION MODEL. LTL PROPERTIES

---

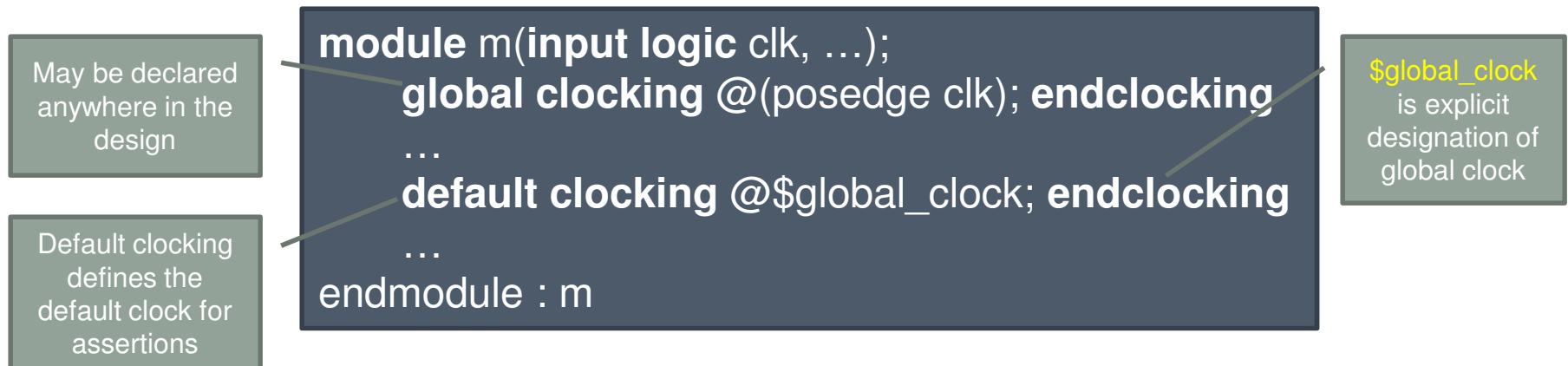
# Linear Time Model. Global Clock

- We assume that the time is linear:
  - There is one global clock (AKA system clock or reference clock) which always ticks (never stops)
  - All signal changes are synchronized with the ticks of the global clock
  - Global clock introduces the notion of discrete time in the system
    - 0, 1, 2, ...
    - Each number corresponds to a tick of the global clock

[Pnu77]

# Global Clock in SystemVerilog

- In simulation the natural notion of global clock are simulation ticks. But such definition makes global clock very expensive
- In SystemVerilog there is a special construct for global clocking definition



**global clocking** is used in simulation, but usually ignored in FV and emulation

In practice most assertions are written relative to some specific clock, not relative to the global clock. Will be discussed later

# Formal Verification Model

- Kripke structure:  $M = \langle Q, I, V, R \rangle$ 
  - $Q$  – Finite set of states
  - $I \subseteq Q$  – Set of initial states
  - $V$  – Set of Boolean variables (labels)
  - $L: Q \rightarrow 2^V$  – Labeling function mapping each state to the set of variables that are true in this state
  - $R \subseteq Q \times Q$  – (Left total) transition relation
    - Shows to which states it is possible to transition from given state

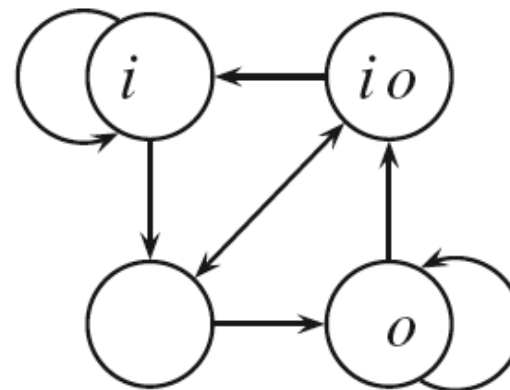
[Kr63]

# Formal Verification Model. Example

```

module m(input logic i, clk, output o);
  wire a = !i;
  global clocking @(posedge clk); endclocking
  always @($global_clock) o <= a;
endmodule : m
  
```

- $V = \{i, o\}$
- $Q = \{\emptyset, \{i\}, \{o\}, \{i, o\}\}$
- $I = Q$
- $R$



# Symbolic Representation

- Each  $x$  variable is represented as a pair:
  - Current value (its value at the current time moment):  $x$
  - Next value (its value at the next time moment):  $x'$
- Each set and relation is represented by its characteristic function
  - E.g.,  $R = i \oplus o'$
- In SystemVerilog there is a notation of next value:
  - $\$future\_gclk(x)$ 
    - E.g.,  $i \oplus o'$  corresponds to  $i \wedge \$future\_gclk(o)$
- Other future sampled value functions:
  - $\$rising\_gclk(x) \Leftrightarrow !x \& \$future\_gclk(x)$  (for bit variables)
  - $\$falling\_gclk(x) \Leftrightarrow x \& !\$future\_gclk(x)$  (for bit variables)
  - $\$steady\_gclk(x) \Leftrightarrow x === \$future\_gclk(x)$
  - $\$changing\_gclk(x) \Leftrightarrow x !== \$future\_gclk(x)$

# Linear Time Model

- Linear time FV model defines a number of legal infinite traces
- Specification language describes the properties of these traces

```

module m(input logic clk, i, output logic o);
  global clocking @(posedge clk); endclocking
  default clocking @$global_clock; endclocking
  always @(posedge clk) o <= !i;
  assert property (i | => !o);
endmodule : m
  
```

Feasible  
traces

i	0	0	1	1	...
o	0	1	1	0	...

i	0	1	0	1	...
o	1	1	0	1	...

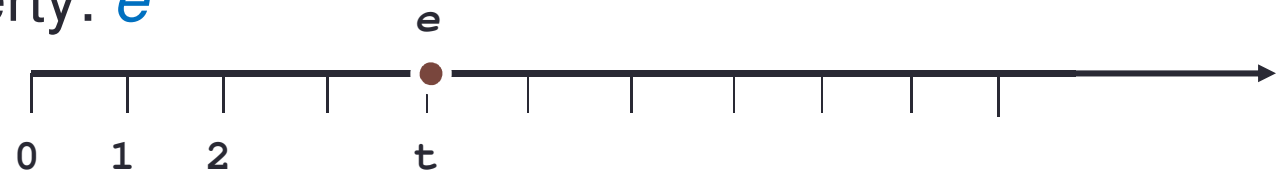
Infeasible  
trace

i	0	0	1	1	...
o	0	0	1	1	...

# Linear Time Language (LTL) Properties

- Properties are temporal statements that can be true or false
  - Properties have a starting point ( $t$ ), but no end point

- Boolean property:  $e$



- Nexttime property:  $\text{nexttime } p$

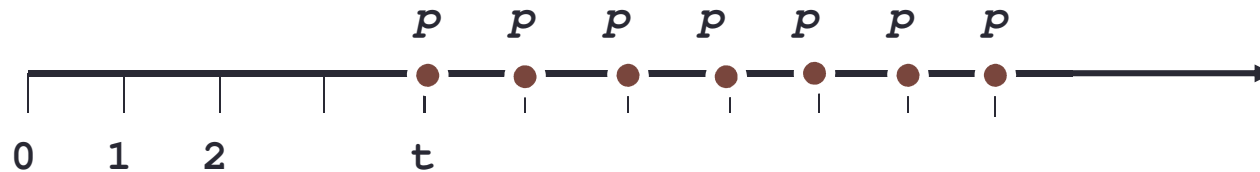


- In SVA there exists also a strong version:  $\text{s\_nexttime}$ 
  - Discussed later

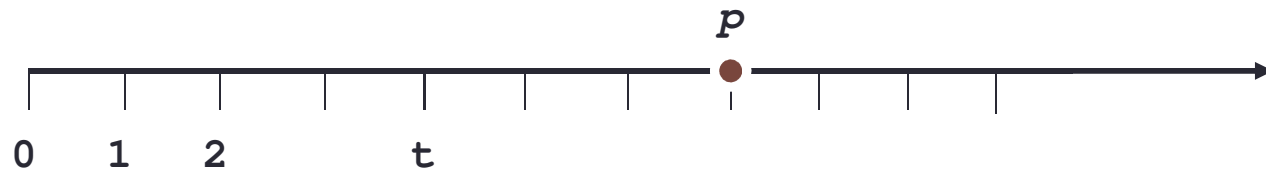


# LTL Properties (cont.)

- Always property: **always  $p$**

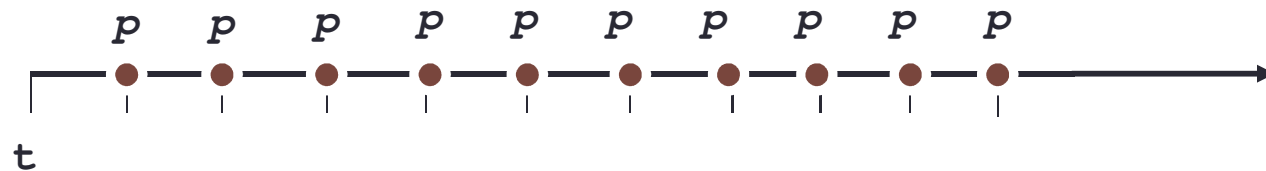


- Eventually property: **s\_eventually  $p$**

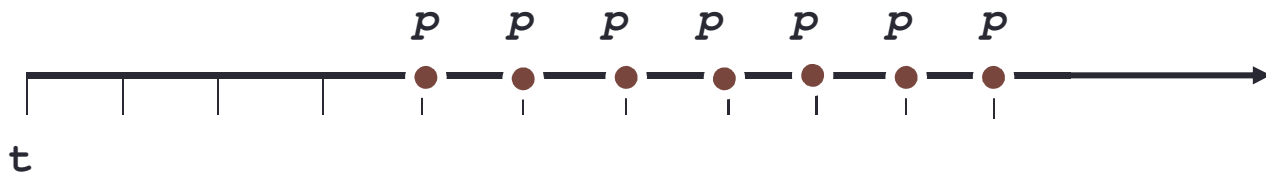


# Compound Properties

- True from next moment: **nexttime always  $p$**



- True from some moment: **s\_eventually always  $p$**



- Infinitely often: **always s\_eventually  $p$**



# Boolean Connectives

- Negation: **not**  $p$
- Conjunction:  $p$  **and**  $q$
- Disjunction:  $p$  **or**  $q$
- Implication:  $p$  **implies**  $q$
- Equivalence:  $p$  **iff**  $q$
- Condition: **if** ( $e$ )  $p$  **else**  $q$
- Case

# Counterexample

- Counterexample – trace witnessing a property failure
- In general case – infinite
- May be finite
  - Meaning that any infinite extension of this finite trace is a counterexample

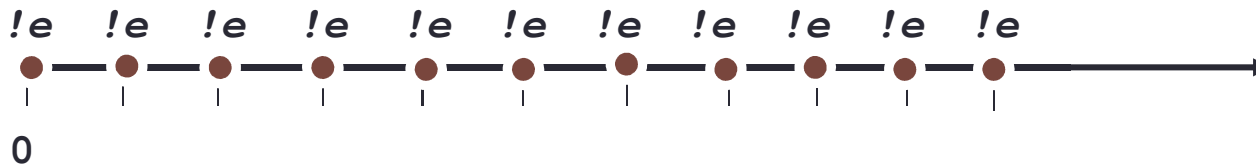
# Safety and Liveness

[AS87, KV01]

- Safety properties
  - All their counterexample are finite
    - E.g., **always  $e$**



- Liveness properties
  - All their counterexamples are infinite
    - E.g., **s\_eventually  $e$**

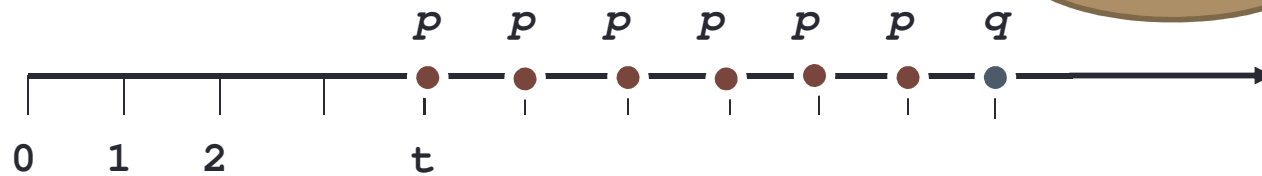


- Hybrid properties also exist
  - Sometimes also called “liveness”

# LTL Properties. Until

- Non-overlapping until

- $p$  until  $q$  – if  $q$  does not happen,  $p$  holds forever
- $p$  s\_until  $q$  –  $q$  must eventually happen

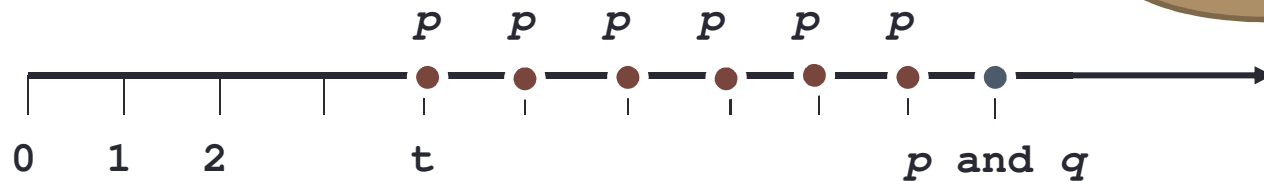


Safety

(General) liveness

- Overlapping until

- $p$  until\_with  $q$  – if  $q$  does not happen,  $p$  holds forever
- $p$  s\_until\_with  $q$  –  $q$  must eventually happen



Safety

(General) liveness

# Bounded Versions

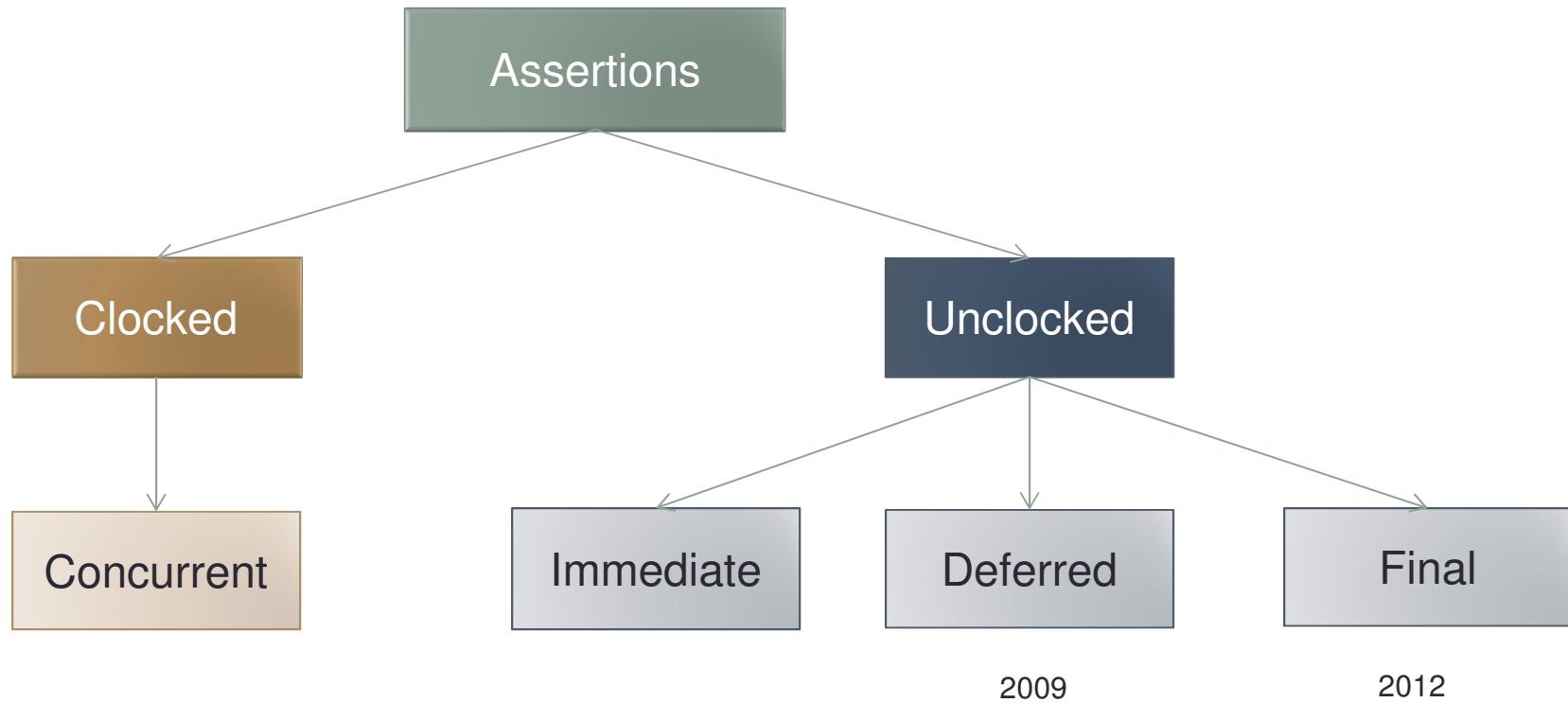
Property	Semantics
$[s\_]\text{nexttime } [m] p$	$[s\_]\text{nexttime } \dots [s\_]\text{nexttime } p$
$[s\_]\text{eventually } [m:n] p$	$[s\_]\text{nexttime } [m] p \text{ or } \dots \text{ or } [s\_]\text{nexttime } [n] p$
$s\_eventually [m:\$] p$	$s\_nexttime [m] s\_eventually p$
$[s\_]\text{always } [m:n] p$	$[s\_]\text{nexttime } [m] p \text{ and } \dots \text{ and } [s\_]\text{nexttime } [m] p$
$\text{always } [m:\$] p$	$\text{nexttime } [m] \text{ always } p$

# ASSERTION STATEMENTS

---



# Assertion Kinds



# (Concurrent) Assertion Statements

- Assertions

- Insure design correctness

**assert property** ( $p$ );

- Assumptions

- Model design environment

**assume property** ( $p$ );

- Cover statements

- To monitor coverage evaluation

**cover property** ( $p$ );

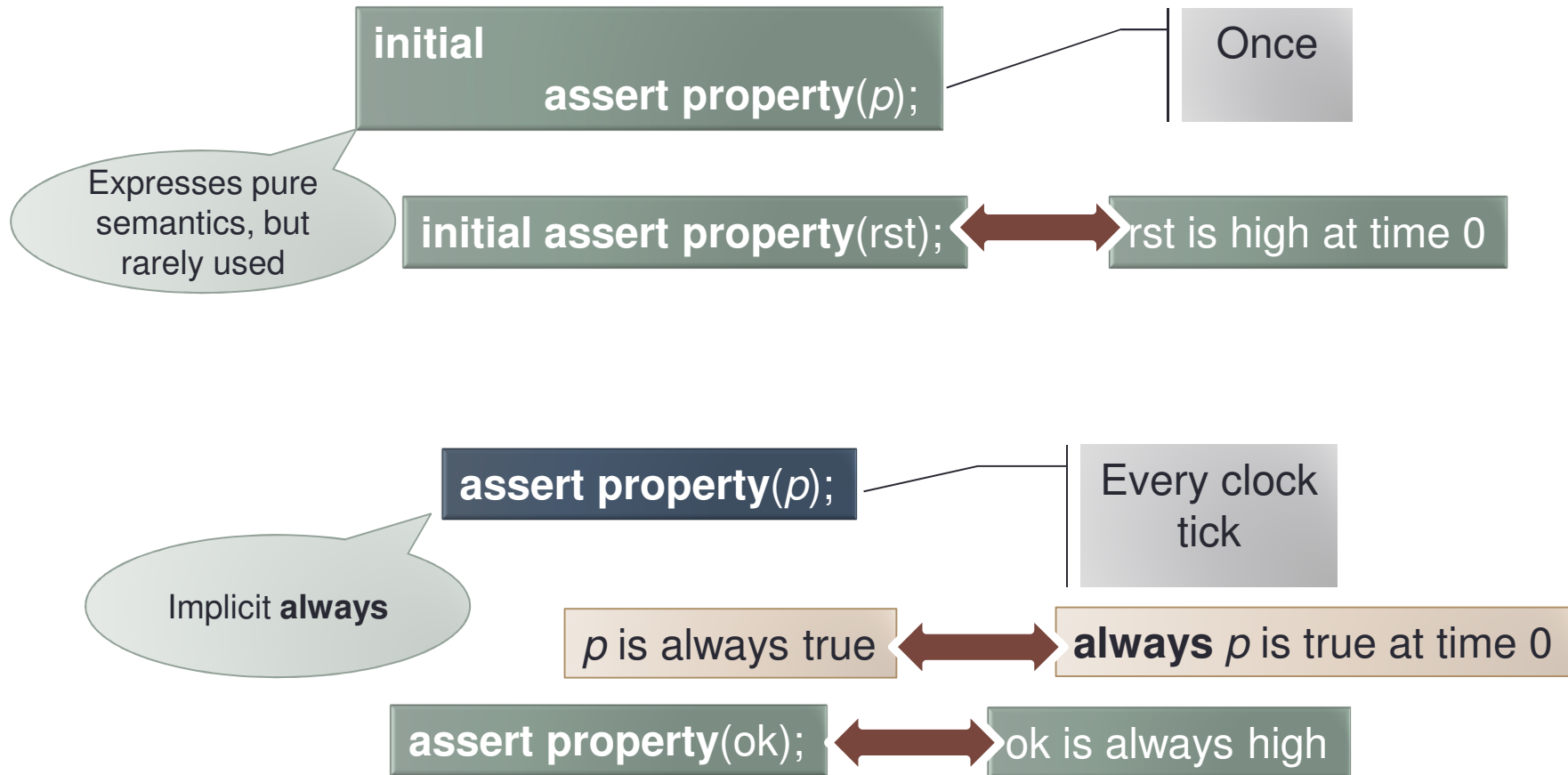
- Restrictions

- To specify formal verification constraint

**restrict property** ( $p$ );

# Assertion Placement

- Inside initial procedure execute only once
- Outside of initial procedure execute continuously



# Assertions

- Specify requirements from DUT
- FV
  - Mathematically proves assertion correctness
- DV
  - Checks assertion correctness for given simulation trace

**initial assert property ( $p$ );**

Passes iff  $p$  is true at time 0 on all feasible traces



Fails iff  $p$  is false at time 0 on some feasible trace

# Assumptions

- Specify requirements from environment
- FV
  - Restricts the set of feasible traces in the model
- DV
  - Checks assertion correctness for given simulation trace

```
assume property (in == !inv_in)
else $error("Inputs in and inv_in are expected to be inverse");
```

- From FV point of view, the DUT acts as an assumption
- Contradictory assumptions (with each other or with the model) cause all assertions **to pass**
  - This is called an *empty model*

# Restrictions

- Specify condition for which FV has been performed
- FV
  - Restricts the model
    - Same as assumption
- DV
  - Completely ignored

```
restrict property (opcode == OP_ADD);
```

# Cover

- Specify scenario you wish to observe
- FV
  - Mathematically prove that the property holds on some feasible trace
- DV
  - Capture scenario in simulation trace

```
cover property (read[*2]);
```

- From FV point of view



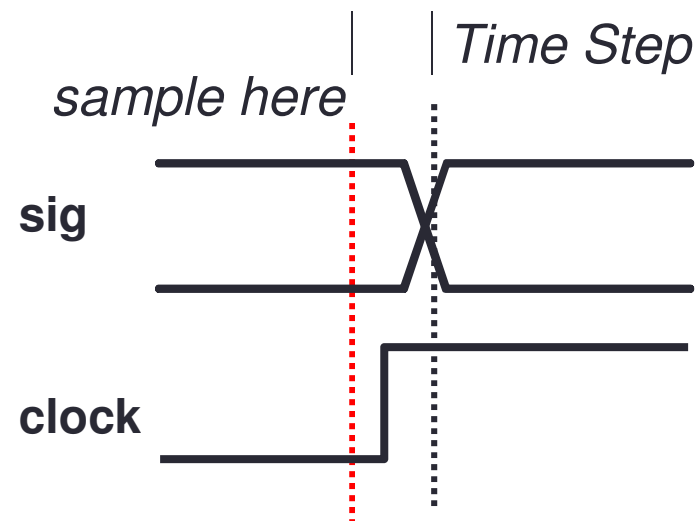
# Formal View on Assertions and Assumptions

- Set of assumptions
  - **initial assume property** ( $q_1$ );
  - ...
  - **initial assume property** ( $q_m$ );
- and assertions
  - **initial assert property** ( $p_1$ );
  - ...
  - **initial assert property** ( $p_n$ );
- is equivalent to the following single assertion
  - **initial**  
**assert property** ( $q_1$  and ... and  $q_m$   
**implies**  $p_1$  and ...  $p_n$ );



# Sampling

- Signal values are sampled at the beginning of simulation tick



# SEQUENCES AND PROPERTIES

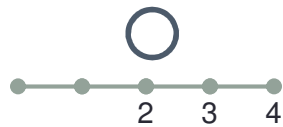
---

# Sequence

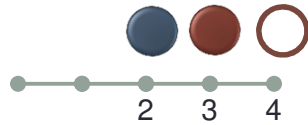
- Informal definition
  - *Sequence* is a rule defining a series of values in time
- A sequence does not have a truth value, it has one initial point and zero or more *match* points
- When a sequence is applied to a specific trace, it defines zero or more *finite* fragments on this trace starting at the sequence initial point and ending in its match points
  - Essentially, sequence is a regular expression

# Example

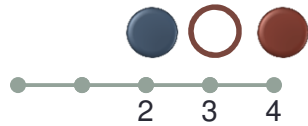
- *Read* is followed by *write* in one or two clock ticks
  - *read* ##[1:2] *write*
  - Let starting point of this sequence be  $t = 2$



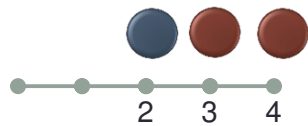
No match



Single match at 3



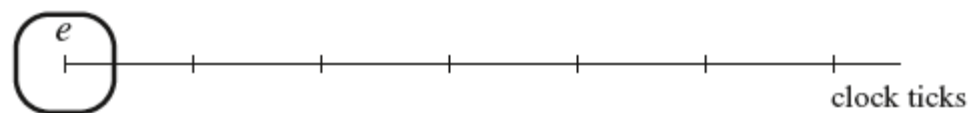
Single match at 4



Two matches (at 3 and 4)

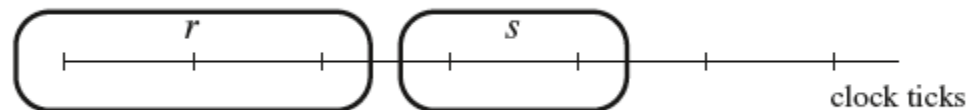
# Boolean Sequence

- Boolean expression  $e$  defines the simplest sequence – a Boolean sequence
  - This sequence has a match at its initial point if  $e$  is true
  - Otherwise, it does not have any satisfaction points at all



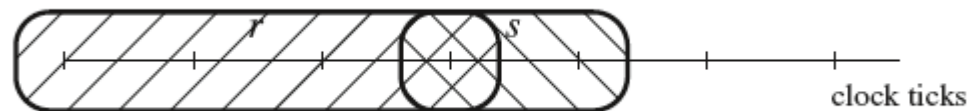
# Sequence Concatenation

- Sequence concatenation:  $r \# \# 1 s$ 
  - There is a match of sequence  $r \# \# 1 s$  if there is a match of sequence  $r$  and there is a match of sequence  $s$  starting from the clock tick immediately following the match of  $r$ 
    - In other words, a finite trace matches  $r \# \# 1 s$  iff it can be split into two adjacent fragments, the first one matching  $r$ , and the second one matching  $s$ .



# Sequence Fusion

- *Sequence fusion*  $r \# \# 0 s$  is an overlapping concatenation
  - The fusion of sequences  $r$  and  $s$ , is matched iff for some match of sequence  $r$  there is a match of sequence  $s$  starting from the clock tick where the match of  $r$  happened



# Zero Repetition (Empty Sequence)

- $s[*0]$ 
  - sequence admitting only an empty match
    - Matches on any trace but the trace fragment is empty (does not contain clock ticks)

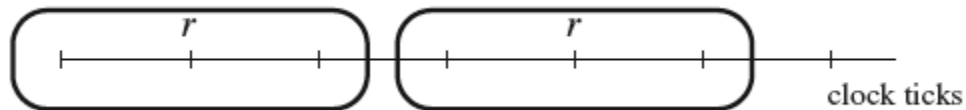


# Sequence Disjunction

- *Sequence disjunction  $r$  or  $s$*  is a sequence which has a match whenever either  $r$  or  $s$  (or both) have a match

# Consecutive Repetition

- Repetition
  - $r[*0]$  is an empty sequence
  - If  $n > 0$  (const.)
    - $r[*n] \Leftrightarrow r[*n-1] \# \# 1 r$



- Finite repetition range
  - $r[*n:n] \Leftrightarrow r[*n]$
  - $r[*m:n] \Leftrightarrow r[*m:n-1] \text{ or } r[*n], m < n$

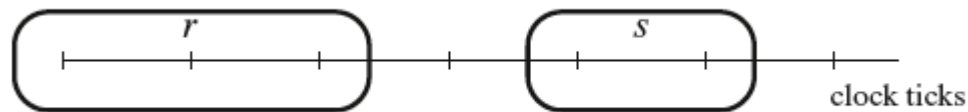
# Infinite Repetition Range

- Infinite range: repeat  $s$   $n$  or more times
- Formal definition
  - Sequence  $s[*1:\$]$  matches trace fragment  $i : j$  if it is possible to divide this trace fragment into one or more consecutive fragments so that each such fragment matches  $s$
  - $s[*0:\$] \Leftrightarrow s[*0] \text{ or } s[*1:\$]$
  - $s[*n:\$] \Leftrightarrow s[*0:n-1] \text{ or } s[*1:\$], n > 1$
- Shortcuts (SVA 2009)
  - $s[*] \Leftrightarrow s[*0:\$]$  – Zero or more times
  - $s[+] \Leftrightarrow s[*1:\$]$  – One or more times

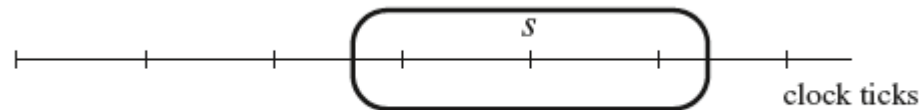
$s[*n:\$]$  does *not* mean that sequence  $s$  is repeated infinitely many times, but that it is repeated  $n$  or more (finite) number of times

# Sequence Concatenation and Delay

- $r \#\#0 s$  is a sequence fusion
- $r \#\#1 s$  is a sequence concatenation
- $r \#\#n s$ , where  $n > 1$  is defined recursively
  - $r \#\#n s \Leftrightarrow r \#\#1 1[*n-1] \#\#1 s$



- $\#\#n s \Leftrightarrow 1[*n] \#\#1 s$



# Delay Ranges

- $r \#\#[0:0] s \Leftrightarrow r \#\#0 s$
- $r \#\#[m:n] s \Leftrightarrow (r \#\#1 \ 1[*m-1:n-1] \#\#1 s)$ , where  $n \geq m > 0$
- $r \#\#[0:n] s \Leftrightarrow (r \#\#0 s) \text{ or } (r \#\#[1:n] s)$ , where  $n > 0$
- $r \#\#[m:\$] s \Leftrightarrow (r \#\#1 \ 1[*m-1:\$] \#\#1 s)$ , where  $m > 0$
- $r \#\#[0:\$] s \Leftrightarrow (r \#\#0 s) \text{ or } (r \#\#[1:\$] s)$ , where  $n > 0$
  
- $\#\#[m:n] s \Leftrightarrow 1 \#\#[m:n] s$ , where  $n \geq m \geq 0$
- $\#\#[m:\$] s \Leftrightarrow 1 \#\#[m:\$] s$ , where  $m \geq 0$
  
- Shortcuts (SVA 2009)
  - $\#\#[*] \Leftrightarrow \#\#[*0:\$]$
  - $\#\#[+] \Leftrightarrow \#\#[*1:\$]$

# Other Sequence Operators

- Go to repetition:  $e[->n]$ ,  $e[->m:n]$ 
  - $e$  is a Boolean
- Non-consecutive repetition:  $e[=n]$ ,  $e[=m:n]$ 
  - $e$  is a Boolean
- Intersection:  $r$  **intersect**  $s$
- Conjunction:  $r$  **and**  $s$
- Containment:  $r$  **within**  $s$
- Throughout:  $e$  **throughout**  $s$
- First match: **first\_match**( $r$ )
- Sequence methods
  - $r$ .triggered
  - $r$ .matched

# Sequential Property

- Strong sequential property
  - **strong(s)** is true in clock tick  $i$  iff sequence  $s$  with initial point  $i$  has a match
  - Sequence  $s$  should not admit an empty match
- Weak sequential property
  - **weak(s)** is true in clock tick  $i$  iff there is no finite trace fragment  $i : j$  witnessing inability of sequence  $s$  with the initial point  $i$  to have a match.
  - Sequence  $s$  should not admit an empty match
- In assertions, assumptions and restrictions **weak** may be omitted
- In cover statements **strong** may be omitted

# Sequential Properties. Examples

- **initial assert property** (rst[\*2]);
  - Same as **initial assert property** (weak(rst[\*2]));
    - For global clock it is also the same as **initial assert property** (strong(rst[\*2]));

## **X**initial assert property (rst[\*]);

- Admits empty match
- **initial assert property** (rst[\*] ##1 ready);
  - Same as **initial assert property** (rst until ready);
- **initial assert property** (strong(rst[\*] ##1 ready));
  - Same as **initial assert property** (rst s\_until ready);
- **initial assert property** (##[\*] ready);
  - Tautology
- **initial assert property** (strong(##[\*] ready));
  - Same as **initial assert property** (s\_eventually ready);



# Suffix Implication

- A *suffix implication* is a property built from a sequence ( $s$ ) and a property ( $p$ )
  - $s$  – *antecedent* – triggering condition
  - $p$  – *consequent* – checked when triggering condition holds
  - Suffix implication is true when its consequent is true upon each completion of its antecedent
- Overlapping implication:  $s \mid\rightarrow p$ 
  - consequent is checked starting from the moment of *every* nonempty match of the antecedent
- Nonoverlapping implication:  $s \mid\Rightarrow p$ 
  - consequent is checked starting from the next clock tick of every match of the antecedent
  - For singly-clocked properties
    - $s \mid\Rightarrow p \Leftrightarrow s \#\#1 \ 1 \mid\rightarrow p$

# Examples

- Request must be granted
  1. **assert property** (req |-> **s\_eventually** gnt);
  2. **assert property** (req |=> **s\_eventually** gnt);
    - Both assertions allow sending one grant to multiple requests
- Request must be granted in 3 cycles
  1. **assert property** (req |-> **##3** gnt); or
  2. **assert property** (req |=> **##2** gnt);
- Request must be active until grant is asserted
  1. **assert property**(req |-> req **until** grant);
  2. **assert property**(req |-> req **until\_with** grant);
  3. **assert property**(req |-> req **s\_until** grant);
  4. **assert property**(req |-> req **s\_until\_with** grant);
- Two consecutive alerts must be followed by reset
  - **assert property** (alert[\*2] |=> reset);

# Vacuity

- What do we check in previous assertions if requests cannot be produced by the model?
- Assertion *holds vacuously* if it is redundant
  - E.g., the previous assertions may be rewritten in this case as **assert property (not req);**
- FV tools provide vacuity check
  - The cost is rather high

[Ar03]

# Suffix Conjunction

- A *suffix conjunction* is a property built from a sequence ( $s$ ) and a property ( $p$ )
  - $s$  – *antecedent* – triggering condition
  - $p$  – *consequent* – checked when triggering condition holds
  - Suffix conjunction is true when its consequent is true upon at least one completion of its antecedent
- Overlapping conjunction:  $s \#-\# p$
- Nonoverlapping conjunction:  $s \#=\# p$
- Example:
  - Reset is initially high and when it becomes low it remains low forever

• initial assert property  $(rst[+] \#\#1 \ !rst \ |=> \ \text{always} \ !rst);$

• initial assert property  $(rst[+] \#\#1 \ !rst \ \#=\# \ \text{always} \ !rst);$

Passes if rst is  
always high

Fails if rst is  
always high

# CLOCKS

---

# Clocks

- Assertion clock should be explicitly written or inferred from the default clocking

```
assert property (@clk p);
```

```
assert property @(posedge clk) p;
```

```
default clocking @(posedge clk); endclocking  
...  
assert property (p);
```

# Clock Rewriting

- Unless clock is not inferred as a system clock (=global clock) by an FV tool, the corresponding property is rewritten
- Examples



```
assert property @(posedge clk) e;
```

```
assert property ($rising_gclk(clk) |-> e);
```



```
assert property @(posedge clk) req ==> gnt);
```

```
assert property (($rising_gclk(clk) & req ##1 $rising_gclk(clk)|-> gnt);
```

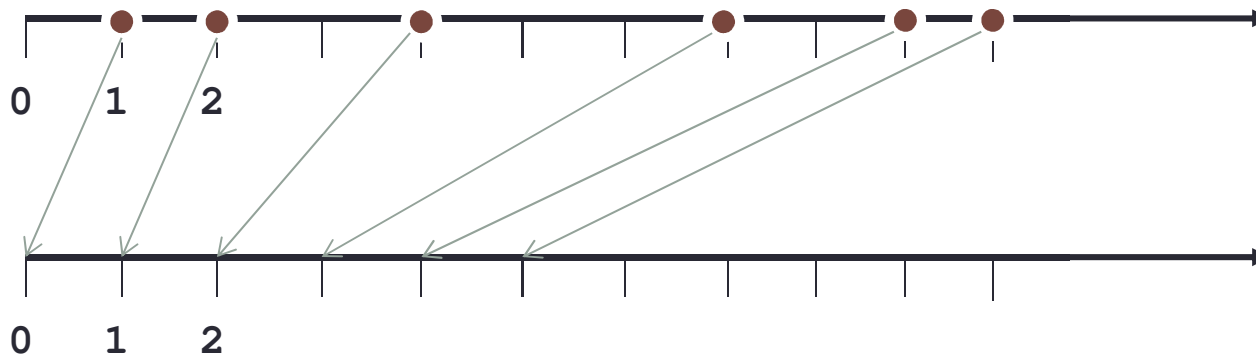
# Clock Fairness

- Clock is *fair* if it ticks infinitely many times
- Without any preliminary knowledge clock fairness is not guaranteed
  - Clock may stop ticking at some moment
- Global clock is fair by its definition



# Clock Fairness (cont.)

- Clock defines a subtrace
  - Only moments corresponding to clock ticks are retained



- When clock is fair the subtrace is infinite
  - Formal semantics does not change
- When clock is not fair the subtrace is finite
  - Need to define property semantics on finite trace

# Weak and Strong Properties

- Weak operators do not require clock to tick
- Strong operators require clock to tick enough times
- Example
  - **nexttime** – weak version
  - **s\_nexttime** – strong version

`initial assert property (@clk nexttime p);`

Passes iff either  $p$  is true at time 1 **or**  $clk$  ticks less than two times

`initial assert property (@clk s_nexttime p);`

Passes iff  $clk$  ticks at least two times **and**  $p$  is true at time 1

# Weak and Strong Properties. Negation

- Negation inverts weakness
  - E.g., **not**, antecedent in **implies**
- Example
  - **not always  $p$**   $\Leftrightarrow$  **s\_ eventually not  $p$**

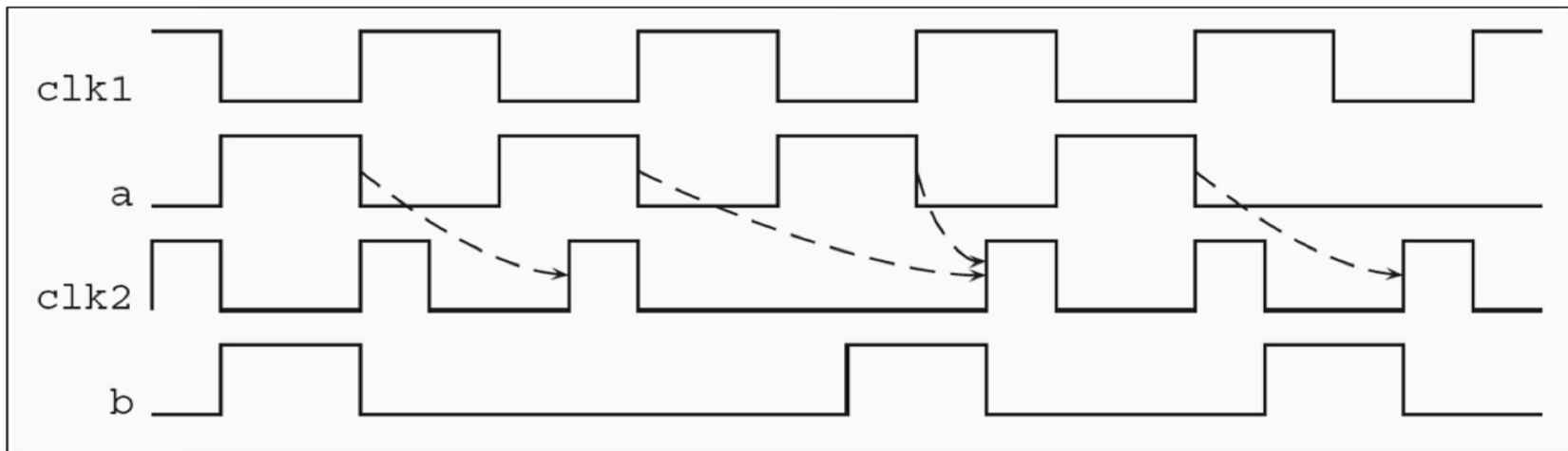
# Mixing Weak and Strong Properties

- Mixing weak and strong properties in most cases is non-intuitive and should be avoided
  - Also for performance reasons
- Example
  - **s\_nexttime always p**
    - Clock should tick at least twice and p should be true at each clock tick starting from time  $t + 1$
  - In some cases mixing is meaningful
    - **s\_eventually always p** – fairness
    - **always s\_eventually p**

# Multiply Clocked Properties

- SVA supports multiply clocked properties

```
assert property(@(posedge clk1) a |=> @(posedge clk2) b);
```



# RESETS

---

# Resets and Aborts

- Reset and abort operators – operators to stop property evaluation when some condition is met
  - Simplify writing assertions in presence of hardware resets
- **disable iff** – main reset of an assertion
- Aborts
  - Asynchronous
    - **accept\_on**
    - **reject\_on**
  - Synchronous
    - **sync\_accept\_on**
    - **sync\_reject\_on**

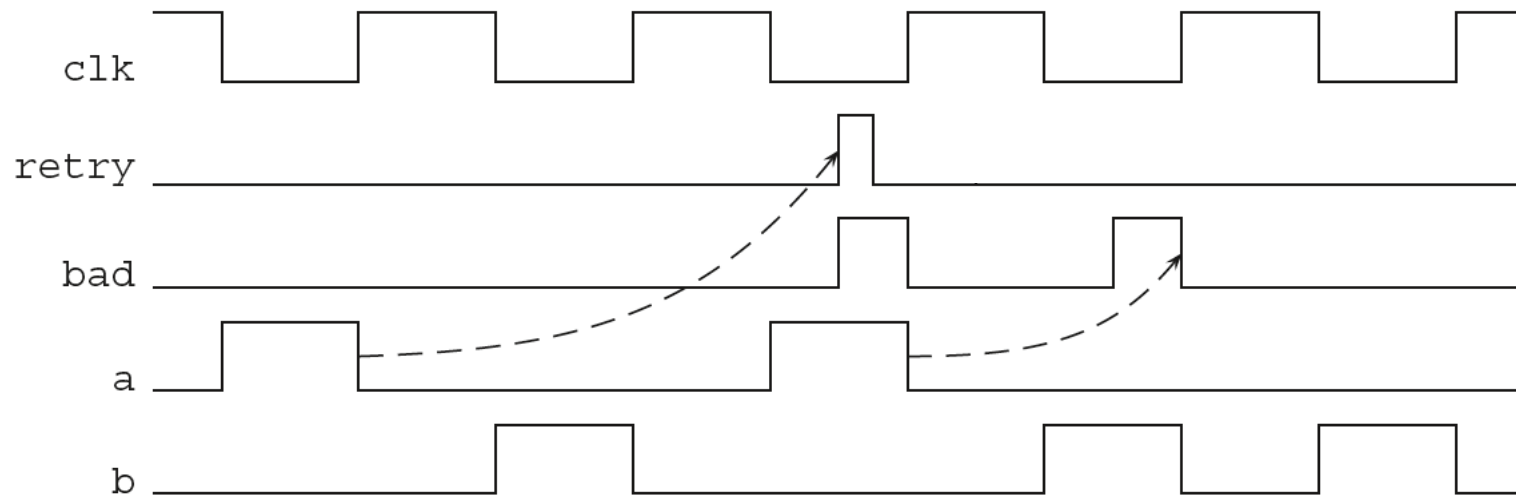
# Aborts

- Asynchronous aborts
  - Ignore the actual clock
    - Checked at each time step
- Synchronous aborts
  - Checked at clock ticks only
- Informal semantics
  - **accept\_on** (*cond*) *p*, **sync\_accept\_on** (*cond*) *p*
    - True if there is no evidence of the failure of *p* before the abort condition has become true
  - **reject\_on** (*cond*) *p*, **sync\_reject\_on** (*cond*) *p*
    - False if there is no evidence of the success of *p* before the abort condition has become true



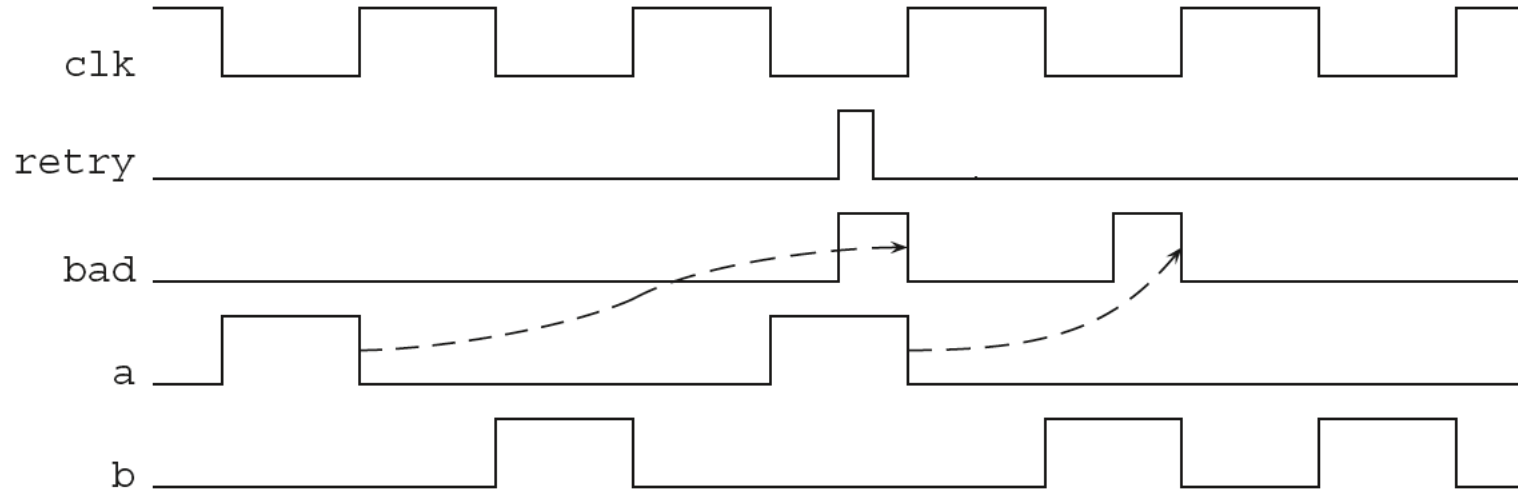
# Asynchronous Aborts. Example

```
assert property(@(posedge clk)
    accept_on (retry) a | => reject_on(bad) b[*2]);
```



# Synchronous Aborts. Example

```
assert property(@(posedge clk)
    sync_accept_on (retry) a |=> sync_reject_on(bad) b[*2]);
```



# One More example

- **reject\_on(rst) 1[\*3]**
  - Property 1[\*3] can never fail
  - Therefore, reject\_on(rst) 1[\*3] fails iff rst becomes high any time during first three clock cycles
- **sync\_reject\_on(rst) 1[\*3]** is equivalent to !rst[\*3]

# Disable Clause

- Syntax
  - **disable iff** (expression)
- Specifies top-level assertion reset
  - At most one in the entire assertion
- In FV may be regarded as **accept\_on** in assertions and assumptions, and as **reject\_on** in cover statements
- Formally introduces a notion of disabled evaluation
  - Evaluation is *disabled* if the assertion evaluation has been aborted because the disable condition became true
- Disable condition is checked continuously, and it is **not** sampled
  - This definition introduces inconsistency between simulation and FV

## Disable Clause (cont.)

- **default disable iff** may be used to specify the default disable condition

```
module m (input logic reset, rst, req, gnt, clk, ...);
    default disable iff reset;

    a1: assert property (@(posedge clk) req | => gnt);
    a2: cover property (@(posedge clk) req ##1 gnt);
    a3: assert property (@(posedge clk) disable iff (1'b0) a | => b);
    a4: assert property (@(posedge clk) disable iff (rst) a | => b);
endmodule : m
```

# ASSERTION SYSTEM FUNCTIONS

---

# Bit-Vector System Functions

Name	Description
\$onehot0	Check that at most one bit in a vector is high
\$onehot	Check that exactly one bit in a vector is high
\$countones	Count number of bits in a vector with value high
\$countbits	Count number of bits having specific value
\$isunknown	Check whether a vector has a bit with value x or z

# Sampled Value Functions

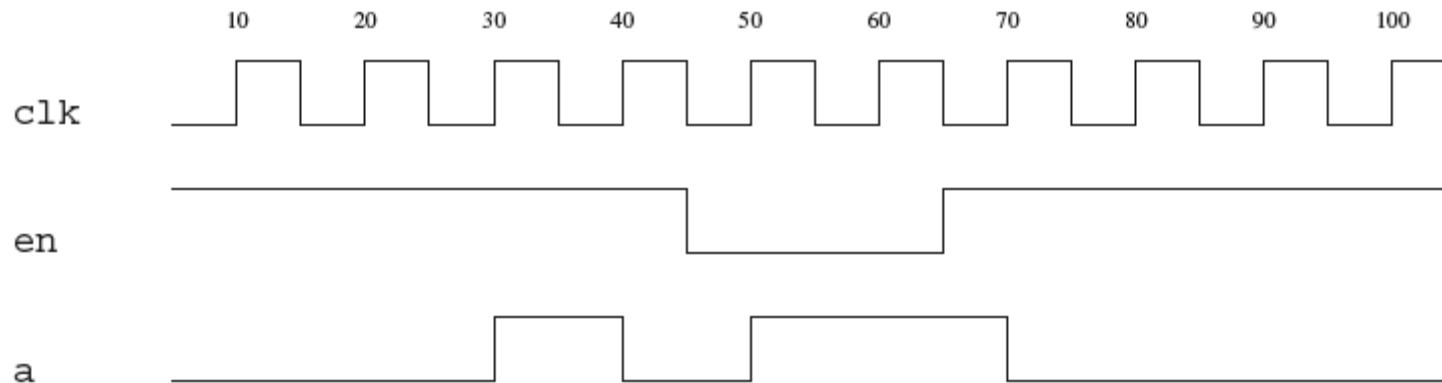
Name	Description
\$sampled	Return sampled value of expression
\$past	Return past value of expression
\$rose	Check whether expression value rose
\$fell	Check whether expression value fell
\$changed	Check whether expression value changed
\$stable	Check whether expression value remained stable



# Past Sampled Values

- `$past(e, n, en, @clk)`
  - `e` – expression
  - `n ≥ 1` – constant expression specifying the number of clock ticks (delay)
  - `en` – gating expression for the clocking event
  - `clk` – clocking event

Time	<code>\$sampled(a)</code>	<code>\$past(a,,,@(posedge clk))</code>
30	0	0
40	1	0
42	0	1
50	0	1
60	1	0
70	1	1
80	0	1
90	0	0



# Values Before Initial Clock Tick

- What happens if for a given time-step there are not enough previous clock ticks?
  - $\$past(e)$  returns an initial value of  $e$
- The initial value of  $e$  is that as computed using the initial values stated in the declaration of the variables involved in  $e$ 
  - If a static variable has no explicit initialization, the default value of the corresponding type is used, **even if the variable is assigned a value in an initial procedure**

FV tools may ignore variable initialization everywhere, except in **checker** constructs. Also, many FV tools consider all variables to be of two-state value type, and therefore they assume that  $\$past(e)$  is 0 in clock tick 0 for any  $e$

# Other Sampled Value Functions

- $\$rose(e, @clk) \Leftrightarrow$   
 $\$past(LSB(e),,,,@clk) !== 1 \ \&\& \ \$sampled(LSB(e)) === 1$
- $\$fell(e, @clk) \Leftrightarrow$   
 $\$past(LSB(e),,,,@clk) !== 0 \ \&\& \ \$sampled(LSB(e)) === 0$
- $\$changed(e, @clk) \Leftrightarrow$   
 $\$past(e,,,@clk) !== \$sampled(e)$
- $\$stable(e, @clk) \Leftrightarrow$   
 $\$past(e,,,@clk) === \$sampled(e)$

# Global Clocking Sampled Value Functions

- May be used only if global clock has is defined
- Past
  - $\$past\_gclk(e) \Leftrightarrow \$past(e, 1, 1, @$global\_clock)$
  - $\$rose\_gclk(e) \Leftrightarrow \$rose(e, @$global\_clock)$
  - $\$fell\_gclk(e) \Leftrightarrow \$fell(e, @$global\_clock)$
  - $\$changed\_gclk(e) \Leftrightarrow \$changed(e, @$global\_clock)$
  - $\$stable\_gclk(e) \Leftrightarrow \$stable(e, @$global\_clock)$
- Future
  - $\$future\_gclk(e)$  – Sampled value of e in the next tick of the global clock
  - $\$rising\_gclk(e) \Leftrightarrow \$sampled(LSB(e)) \neq 1 \ \&\& \ \$future\_gclk(LSB(e)) \equiv 1$
  - $\$falling\_gclk(e) \Leftrightarrow \$sampled(LSB(e)) \neq 0 \ \&\& \ \$future\_gclk(LSB(e)) \equiv 0$
  - $\$changing\_gclk(e) \Leftrightarrow \$sampled(e) \neq \$future\_gclk(e)$
  - $\$steady\_gclk(e) \Leftrightarrow \$sampled(e) \equiv \$future\_gclk(e)$
  - Cannot be nested or used in reset conditions

# METALANGUAGE

---

# Let Declaration

```
let identifier [(port, port, ...)] = expression;
```

- “Compile-time macro” for integral expressions
- Follow normal scoping rules
- Formal arguments may be typed or untyped
- Formal arguments can have default actual arguments
- May be included in a package
- May easily be processed by tools for linting and statistical reporting
  
- Typical usage
  - Templates for Boolean assertions
  - Instrumental code
  - Does not introduce new variables
  - Visible to debuggers

# Let Example

```
module m (input logic clk, rst, ...);  
  logic mod1, mod2;  
  logic req1, req2;  
  logic rsp;  
  let req = mod1 & req1 | mod2 & req2;  
  let gnt = $changed(rsp);  
  ...  
  a: assert property (@(posedge clk) disable iff (rst) req ==> gnt);  
endmodule : check
```

# Sequence and Property Declaration

```
module m(input logic clk, rst, ...);  
  logic my_req;  
  logic gnt;  
  sequence two_reqs(req);  
    req[*2];  
  endsequence  
  property delayed_grant(int n);  
    nexttime [n] gnt;  
  endproperty  
  ...  
  req_granted: assert property (@(posedge clk) disable iff (rst)  
    two_reqs(my_req) |-> delayed_grant(2);  
endmodule
```



# CHECKERS

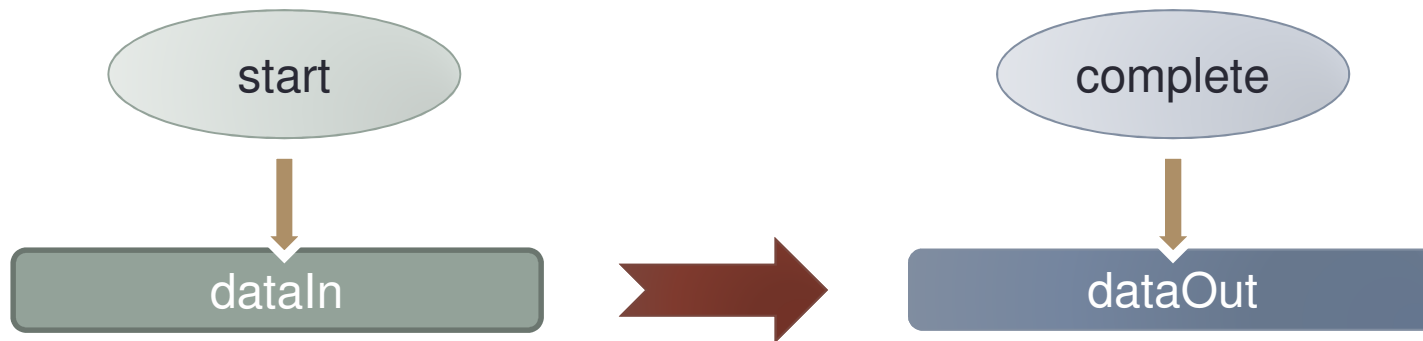
---

# Checkers

- Checkers are SystemVerilog containers to package verification code
  - Both assertions and modeling
- Checker is a kind of hybrid of module, property and assertion
  - May contain (almost) the same constructs as a module
  - Is instantiated as a property (in place)
  - Placed as an assertion
    - Acts as an assertion with complex internal implementation

# Example. Sequential Protocol

- Whenever *start* is high, *dataIn* is valid.
- Whenever *complete* is high, *dataOut* is valid.
- If *start* is high, then the value of *dataIn* at that time must equal the value of *dataOut* at the next strictly subsequent cycle in which *complete* is high
- If *start* is high, then *start* must be low in the next cycle and remain low until after the next strictly subsequent cycle in which *complete* is high
- *complete* may not be high unless *start* was high in a preceding cycle and *complete* was not high in any of the intervening cycles



# Sequential Protocol Verification Checker

```
checker seq_protocol (start, complete, dataIn, dataOut, event clk);  
  
  default clocking @clk; endclocking  
  var type(dataIn) data;  
  
  property match (first, last);   first |=> !first until_with last;   endproperty  
  
  always_ff @clk      if (start) data <= dataIn;  
  
  a_data_check: assert property (complete |-> dataOut == data);  
  a_no_start:  assert property (match(start, complete));  
  a_no_complete: assert property (match(complete, start));  
  
  initial  
    a_initial_no_complete: assert property (!complete throughout start[->1]);  
endchecker : seq_protocol
```

# Checker Binding

```

module top;
logic clock, snda, sndb, sndc, rcva, rcvb, rcvc;
...
trans ta (clock, snda, rcva);
trans tb (clock, sndb, rcvb);
trans #(2) tc (clock, sndc, rcvc);
endmodule : top

```

```

checker eventually_granted (req, gnt, ...);
...
endchecker : eventually_granted

checker request_granted (req, gnt, n, ...);
...
endchecker : request_granted

```

```

module trans #(DEL=1) (input logic clock, in, output logic out);
if (DEL == 1) begin : b
    always @(posedge clock) out <= in;
end
else begin : b
    logic [DEL - 2: 0] tmp;
    always @(posedge clock) begin
        tmp[0] <= in;
        for (int i = 1; i < DEL - 1; i++) tmp[i] <= tmp[i-1];
        out <= tmp[DEL - 2];
    end
end
endmodule : trans

```

```

bind trans eventually_granted check_in2out(in, out, posedge clock);
bind trans: ta, tb request_granted delay1(in, out, , posedge clock);
bind trans: tc request_granted delay2(in, out, 2, posedge clock);

```

# Free Variables

- Checker may have free variables
  - Behave non-deterministically (like free or constrained inputs)
    - FV: consider all possible values imposed by assumptions and assignments
    - Simulation: their values are randomized
  - Free variable values are never sampled
- Limitations
  - Continuous and blocking assignments to free variables are illegal

```
rand bit r;  
bit [2:0] x;  
...  
assign x = r ? 3'd3 : 3'd5;
```

```
rand bit a;  
always_ff @clk a <= !a;
```


# Rigid Variables

- Rigid variables = constant free variables

```
checker data_consistency (start, complete, dataIn, dataOut,  
    event clk, untyped rst);
```

```
    default clocking @clk; endclocking  
    default disable iff rst;
```

```
    rand const type(dataIn) data;
```



Rigid  
variable

```
    a1: assert property (start && data == dataIn ##1 complete[->1]  
        |-> dataOut == data);
```

```
endchecker : data_consistency
```

# Modular Assertion Modeling

```
checker check_fsm(logic [1:0] state, event clk);  
    logic [1:0] astate = IDLE; // Abstract state  
    model_fsm c1(state, clk, astate);  
    check_assertions c2(state, astate, clk);  
endchecker
```

```
checker model_fsm(input event clk, output logic [1:0] astate = IDLE);  
    always @clk  
        case (astate)  
            IDLE: astate <= ...;  
            ...  
            default: astate <= ERR;  
        endcase  
endchecker
```

```
checker check_assertions(state, astate, event clk);  
    default clocking @clk; endclocking  
    a1: assert property (astate == IDLE <-> state inside {IDLE1, IDLE2});  
    //...  
endchecker
```



# Implementing Formal Verification Environment With Checkers

In simulation module input signals are randomized remaining mutually exclusive

```
checker env(event clk, output logic out1, out2);  
  rand bit a, b;  
  m: assume property (@clk $onehot0({a, b}));  
  assign out1 = a;  
  assign out2 = b;  
endchecker : env
```

```
module m(input logic in1, in2, clock, output ...);  
  ...  
endmodule : m
```

```
module top();  
  logic clock, n1, n2;  
  ...  
  m m1(n1, n2, clock, ...);  
  evn env1(clock, n1, n2);  
endmodule : top
```


# LOCAL VARIABLES

---

# Informal Definition

- *Local variable* is a variable associated with an evaluation attempt
  - Local variables are not sampled

```
checker data_consistency (start, complete, dataIn, dataOut,  
    event clk, untyped rst);  
  
    default clocking @clk; endclocking  
    default disable iff rst;  
  
    property p_data_check;  
        var type(dataIn) data;  
        (start, data = dataIn) ##1 complete[->1] |-> dataOut == data;  
    endproperty : p_data_check  
  
    a1: assert property (p_data_check);  
  
endchecker : data_consistency
```



Local variable

Match item

# Example

- Check that the value of *dataIn* when *start* is high coincides with the value of *dataOut* read in *n* clock ticks
- If  $n = \text{const}$

```
assert property (start |-> ##n dataOut == $past(dataIn, n) );
```

- If  $n$  is not const

```
property data_check;  
  var type(n) ctr;  
  (start, ctr = n - 1) ##1 (ctr > 0, ctr--)[*] ##1 (ctr == 0)  
  |-> dataOut = dataIn;  
endproperty : data_check  
  
assert property (data_check);
```

# Local vs. Rigid Variables

- Local variables are “broader” than rigid variables
  - They are mutable
- Local variables are more intuitive
- Local variables are supported in simulation, rigid variables are not
- Rigid variables are FV friendly – their implementation is straightforward
  - Efficient implementation of local variables in FV is challenging

# RECURSIVE PROPERTIES

---

# Recursive Properties

- Properties may be recursive

```
property prop_always (p) ;  
    p and nexttime prop_always (p);  
endproperty
```

```
initial assert property (@(posedge clk) prop_always (ok));
```

```
property prop_weak_until (p, q);  
    q or (p and nexttime prop_weak_until (p, q));  
endproperty
```

```
initial assert property (@(posedge clk) prop_weak_until (req, gnt));
```

# EFFICIENCY AND METHODOLOGY TIPS

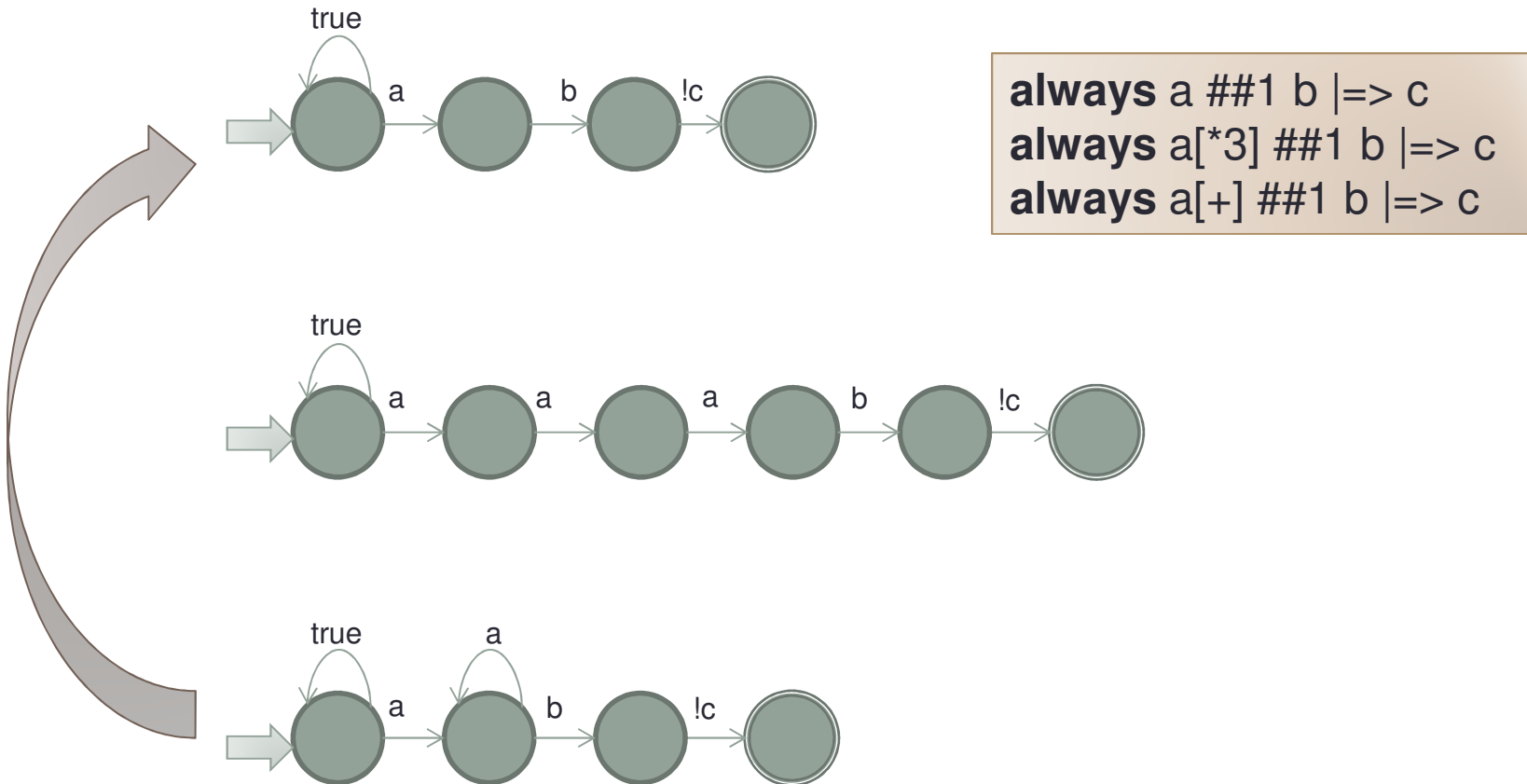
---



# Assertion Compilation for FV

- Assertions are usually compiled into finite automata [Var96]
  - Typical for FV and emulation
  - Sometimes done for simulation as well
- Safety assertions are compiled into (conventional) finite automata on finite words
- Liveness and hybrid assertions are compiled into finite automata on infinite words (e.g., Büchi automata):
  - Finite automata on finite words + fairness conditions
- Complexity of automaton reflects well FV efficiency
- Another factor is the number of state variables in the model

# Automaton-Based Compilation. Example



# Avoid Large and Distant Time Windows



```
assert property (start ##1 !complete[0:1000] ##1 complete | => done);
```



```
assert property (start ##1 complete[->1] | => done);
```



```
assert property (start ##1 !complete[*] ##1 complete | => done);
```

Also applies to bounded property operators and \$past

# Avoid Using Liveness Assertions Unless Really Needed

Request must be active until grant is asserted

✗

```
assert property(req |-> req s_until grant);
```

✓

```
assert property(req |-> req until grant);
```

Clock is fair

✓

```
assert property (s_eventually clk);
```

✗

```
assert property (req |-> ##[1:1000] gnt);
```

✓




```
assert property (req |-> s_eventually gnt);
```




Do you really want to check that grant is eventually received?

Strong operators have clumsier syntax to prevent inadvertent usage


Liveness assertion is usually better than a safety assertion with a large time window

# Avoid Mixing Weak and Strong Properties

 **s\_nexttime always p**  **nexttime always p** 

 **nexttime s\_eventually p**  **s\_nexttime s\_eventually p** 

Sometimes this is unavoidable

**always s\_eventually p** 

# Past vs. Future

- Future value functions are cheap in FV
  - Recall that each variable is represented as a pair (v, v')
- Past value functions are more expensive
  - They introduce new flip-flops (=variables)
- Need to optimize the usage of \$past

```
logic check;  
logic [31:0] a, b, c;
```



```
assert property (##1 check |-> $past(c) == $past(a) + $past(b));
```



```
assert property (##1 check |-> $past(c == a + b));
```

# Intersection Family

- Sequence operators from *intersection* family (**intersect**, **and**, **within**) are expensive
  - These operators are not inefficient by themselves, but allow to concisely code complex sequences
    - **Use only where appropriate**

Each transaction should contain two read requests and three write requests

Common case

```
assert property (start |-> read[=2] intersect write[=3] intersect complete[->1]);
```

If known that all reads come after writes

```
assert property (start |-> write[=3] ##1 read[=2] intersect complete[->1]);
```

- Top-level conjunction in a sequence promoted to property is not expensive

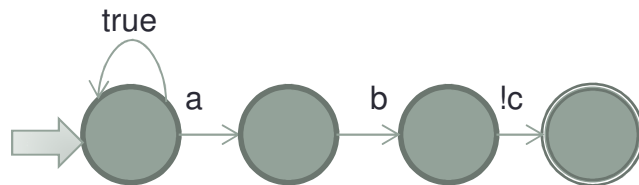
```
assert property (en |-> r and s);
```

```
assert property (en |-> (r and s) ##1 a);
```

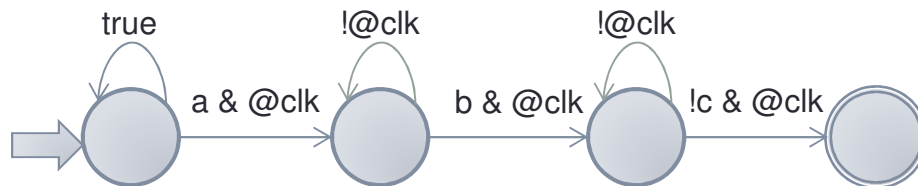
This is rather efficient

# Assertion Clock

- Assertions governed by a global clock are more efficient in FV than assertions governed by an arbitrary clock



**always a ##1 b | => c**



**@clk always a ##1 b | => c**

FV tools may automatically infer the global clock from a singly clocked design



# Local vs. Free and Rigid Variables

- Implementing free and rigid variables is straightforward in FV
- Implementing local variables is tricky
- Important advantage of local variables
  - Allow checking assertions in simulation
- Both are usually expensive
- Efficiency is strongly dependent on FV tool
  - May need to experiment

# Assignments vs. Assumptions

- Assignments are usually more efficient than assumptions
  - They add directionality to the search
- Compare

```
assign x = !y;
```

vs.

```
assume property (x != y);
```

# Overlapping vs. Nonoverlapping Transactions

- Nonoverlapping transactions may be modeled deterministically
- Overlapping transactions usually require nondeterministic modeling
  - E.g., local or rigid variables
- Compare:
  - Sequential protocol vs.

```
property p_data_check;  
    var type(dataIn) data;  
    (start, data = dataIn) ##1 complete[->1] |->  
    dataOut == data;  
endproperty : p_data_check
```

If it is known that transactions cannot overlap, model them as nonoverlapping

# Be More Specific

- Being more specific usually pays off. Check only what you really need
  - Don't check for eventuality unless this is essential
  - If events are ordered in a sequence specify this order explicitly
  - If you know that transactions do not overlap model this fact explicitly
- Nulla regula sine exceptione
  - Liveness assertions are usually more efficient than safety assertions with large/distant time windows
  - Being more specific comes at a price of generality
    - However, generality does not help if performance problems prevent us from getting result

# Efficiency: Simulation vs. FV

- Simulation and FV efficiency requirements may be inconsistent
  - Especially when assertion simulation has a transaction-based implementation
- E.g.
  - Infinite ranges and repetitions are efficient in FV, but not in simulation
  - Sequence intersection is efficient in simulation, but not in FV
  - Liveness does not cost in simulation
  - Future value functions are more efficient than past value functions in FV. The situation with simulation is opposite
  - Local variables are rather efficient in simulation, but not in FV

# FUTURE DIRECTIONS AND CHALLENGES

---

# Convergence Between SVA and SVTB

- Coverage features are divided between SVA and SVTB
  - Assertion coverage belongs to SVA
  - Covergroups belong to SVTB
- Currently there is no organic connection between the two
  - Syntax and semantics are absolutely different
- One can consider temporal coverage specification by integrating sequences and covergroups

# Standard Packages

- SVA provides basic assertion capabilities and some sugaring
  - There are many useful properties and functions that could be reused, but are not a basic part of the language
  - It makes sense to standardize them by creating standard property packages
- PSL has some of such common properties embedded into the language, e.g., never, before, next\_event, etc.



# Assertion Libraries

- Using SVA checker mechanism it is possible to create a powerful and flexible standard assertion library with concise library assertion invocation
  - Kind of “next generation” OVL

# AMS Assertions

- AMS = Analog and Mixed Signals
- The intention is to merge SystemVerilog and Verilog-AMS
  - This includes development of AMS assertions and including them into SVA
  - The initial step was done in SV2012: real type support in SVA
    - No continuous time support yet

# TLM Assertions

- SVA covers RTL assertions
- TLM assertions are different
  - Unclocked or approximately clocked
- SVA is too low level for TLM
- Need to extend SVA to cover TLM needs

# Checkers for UVM

- UVM – Universal Verification Methodology
  - Widely used in verification
- Includes verification level – monitors – to check design correctness
  - Part of TB
  - Uses completely different mechanism, does not explore the strength of assertions
    - Implemented as class methods
- Challenge
  - Checkers currently cannot be instantiated in classes
  - Need to enhance them to allow their usage in UVM

# BIBLIOGRAPHY

---

- [Ar03] - R.Armoni et al., *Enhanced Vacuity Detection in Linear Temporal Logic*, 2003
- [AS87] - B. Alpern, F.B. Schneider, *Recognizing safety and liveness*, 1987
- [Kr63] - S. Kripke. *Semantical Considerations on Modal Logic*, 1963
- [KV01] - O. Kupferman, M.Y. Vardi, *Model checking of safety properties*, 2001
- [Pnu77] - A. Pnueli. *The temporal logic of programs*, 1977
- [PSL10] - *IEEE Standard for Property Specification language (PSL)*, 2010
- [SV12] - *IEEE Standard for SystemVerilog*, 2012
- [Var96] - M. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, 1996