Specification Formalisms

CS60030 FORMAL SYSTEMS

PALLAB DASGUPTA,
FNAE, FASc,
A K Singh Distinguished Professor in AI,
Dept of Computer Science & Engineering
Indian Institute of Technology Kharagpur
Email: pallab@cse.iitkgp.ac.in
Web: http://cse.iitkgp.ac.in/~pallab

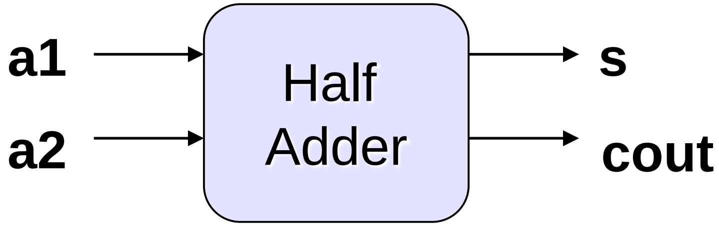
INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

FMSAFE
FORMAL METHODS FOR SAFETY CRITICAL SYSTEMS

Why do we need “temporal” logic?

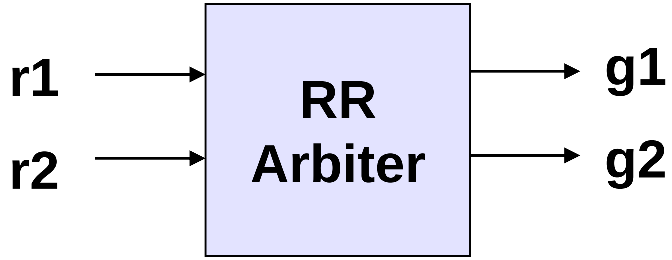
Propositional Logic

- *Boolean formulas*



$cout \Leftrightarrow a1 \wedge a2$

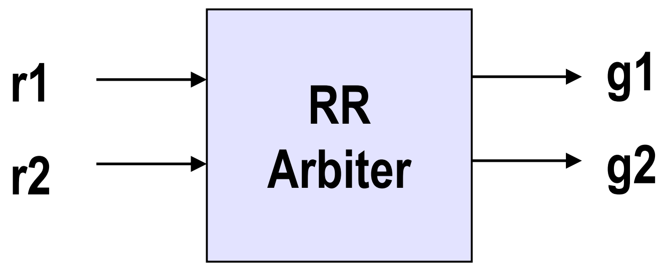
$s \Leftrightarrow a1 \oplus a2$



Temporal Logic

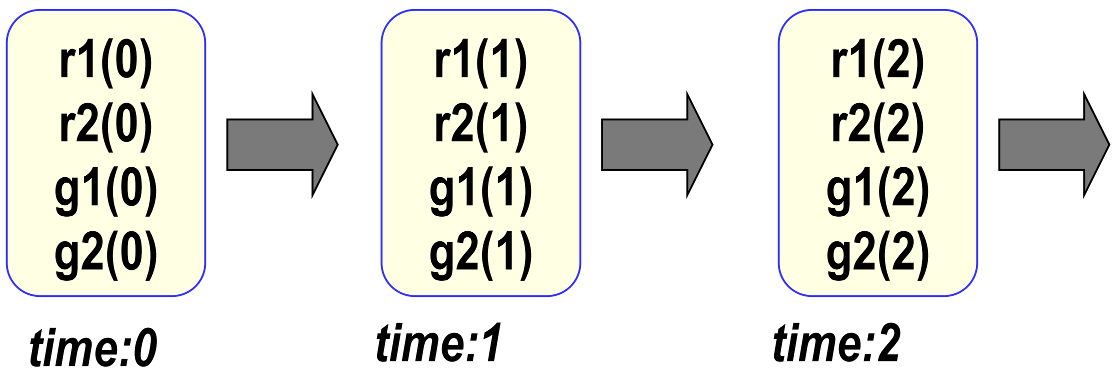
- Properties span across cycle boundaries
- Consider a property of a two way round-robin arbiter
    - *If the request bit r1 is true in a cycle then the grant bit g1 has to be true within the next two cycles*

What does "temporal" mean?



If r1 is true in a cycle then g1 has to be true within the next two cycles

<u>Temporal worlds</u>



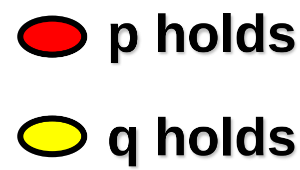
$\forall t \; [\; r1(t) \Rightarrow g1(t+1) \vee g1(t+2) \;]$

In *propositional* temporal logic, the time variable t is implicit.

- For example, we may write:

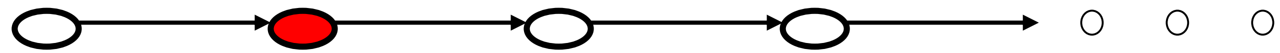
always r1 → (next g1) or (next next g1)

Temporal Operators



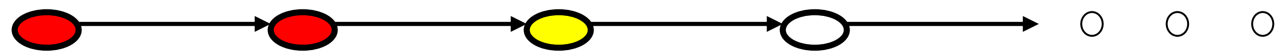
Two fundamental path operators:

- **Next operator** **X p**



  - Xp – *property p holds in the next state*

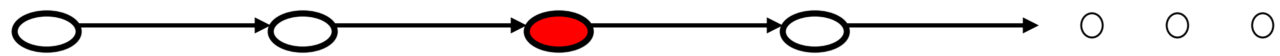
- **Until operator** **p U q**



  - p U q – *property p holds in all states up to the state where property q holds*

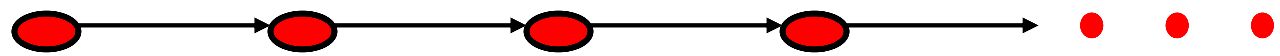
Several derived (and commonly used operators)

- **Eventual operator** **F p**



  - Fp – *property p holds eventually (at some future state)*

- **Always operator** **G p**



  - Gp – *property p holds always (at all states)*

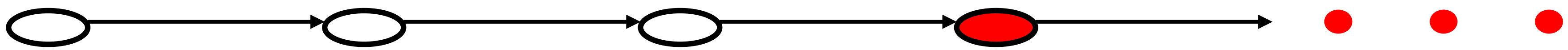
Duality of Always & Eventual Operators:
$\neg Fp = G(\neg p)$ and $\neg Gp = F(\neg p)$

Temporal logics also support all the Boolean operators

All these operators are interpreted over paths of the underlying state machine (Kripke structure)

Nesting of Temporal Operators

F G p



Along the path there exists a state from which *p* will hold forever

G F p

Along the path for all states there will eventually be some state where *p* holds

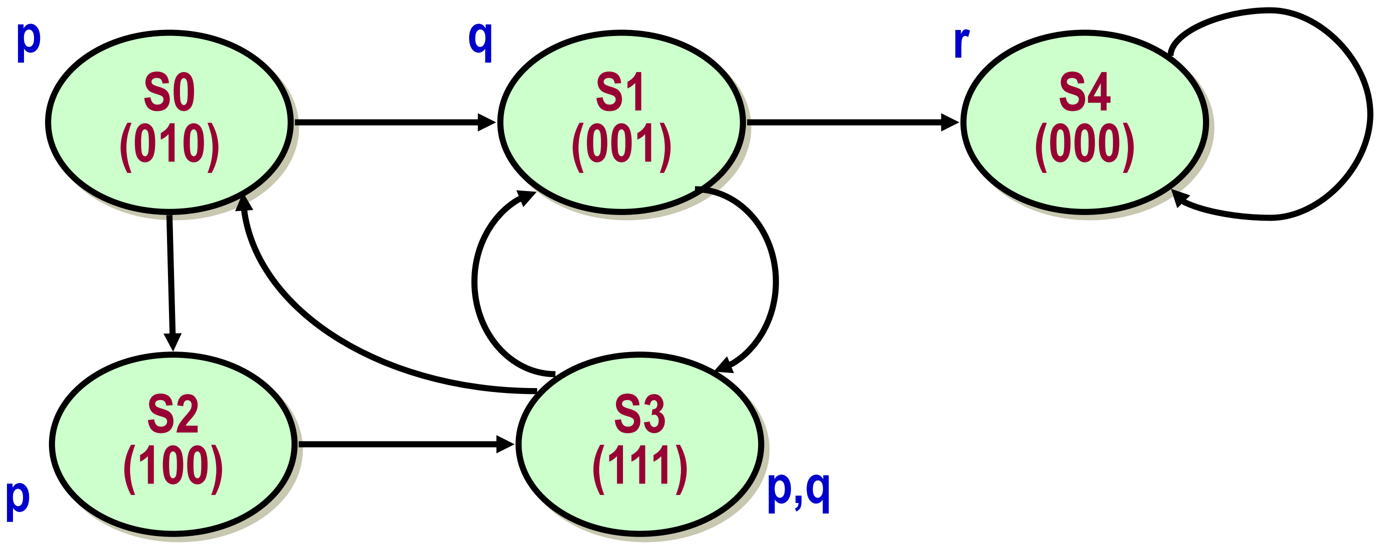
alternatively

Along the path p will hold *infinitely often*

Transition Systems (Kripke Structure)

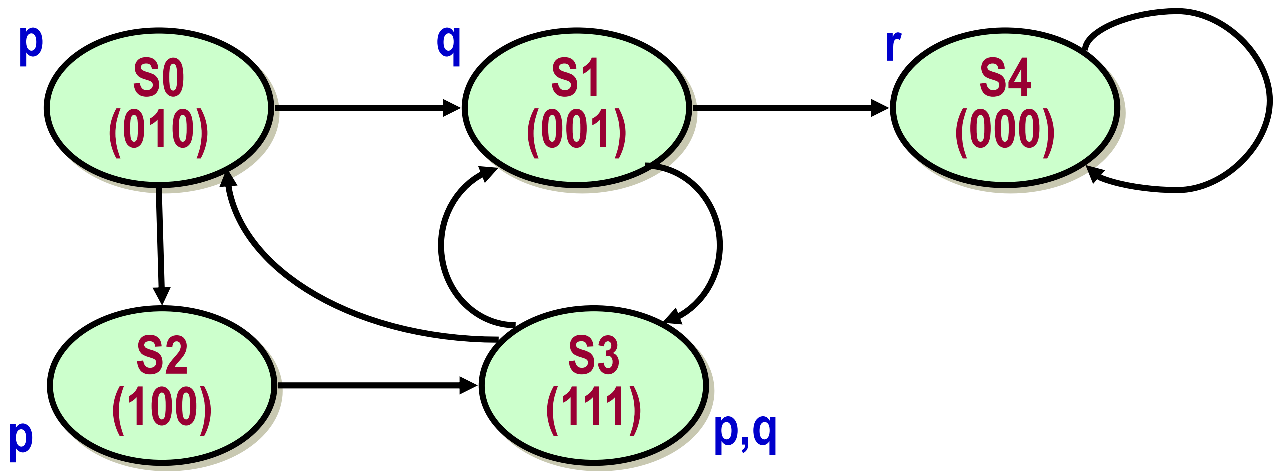
K = (AP, S, S_0, T, L)

- AP is a set of atomic propositions
- S is a set of states
- S_0 is a set of initial states
- $T \subseteq S \times S$, is a *total* transition relation
- L: S $\rightarrow 2^{AP}$ is a labeling function



Path

A path π = n0, n1, … in a Kripke structure, K = (AP, S, S_0, T, L), is a sequence of states such that $\forall k, (n_k, n_{k+1}) \in T$



π^k – suffix of n_k in π

$\pi = n_0, n_1, \ldots, n_k, n_{k+1}, \ldots$

prefix of n_k in π

Sample paths:

s0, s1, s4, s4, s4, …

s0, s2, s3, s0, s2, s3, …

s0, s2, s3, s1, s3, s0, …

Linear Temporal Logic (LTL)

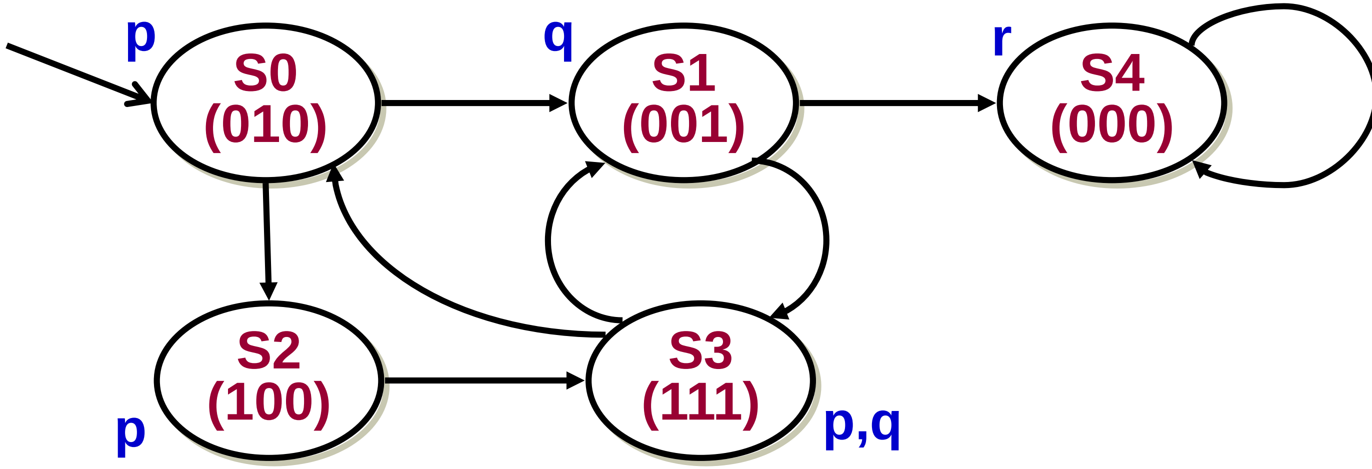
Syntax:

- Given a set, AP, of atomic propositions:
  - All Boolean formulas over AP are LTL properties, and
  - If *f* and *g* are LTL properties, then so are ¬f, X f, and f U g

Semantics:

- A Kripke structure K models a LTL property g (denoted as K |= g) iff for every path π, which starts at some initial state of K, π |= g
- This means that the property does not hold on K if there is any path in K which refutes the property

Examples



The property pUq holds

The property Fq holds

The property GFq does not hold

- Counterexample trace: s0, s1, s4, s4*

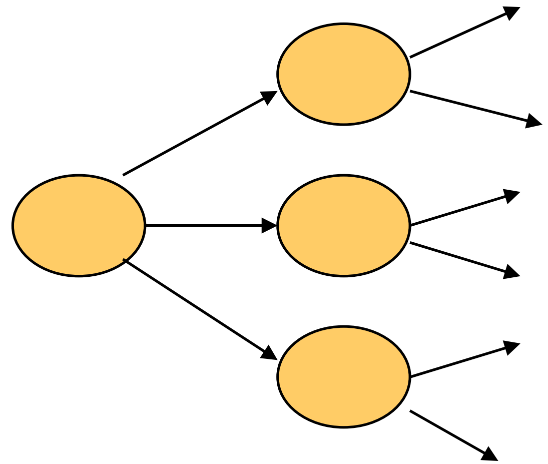
The property p U (qUr) does not hold

- Counterexample trace: s0, s2, s3, s0, (s2, s3, s0)*

Path Quantifiers

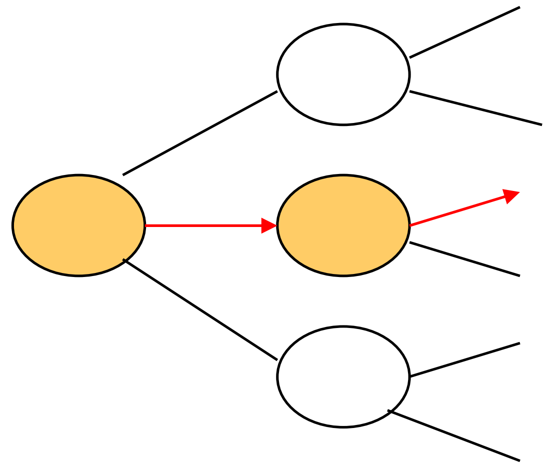
A

“ for all paths … ”



E

“ there exists a path … ”



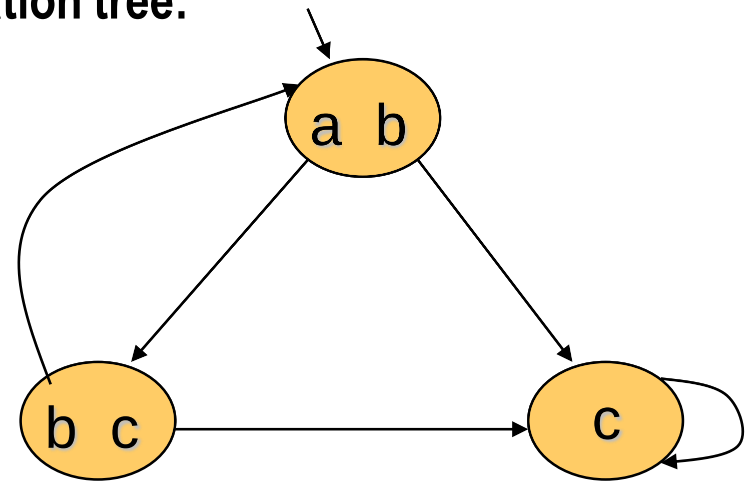
Used to specify that all of the paths or some of the paths starting at a particular state have some property

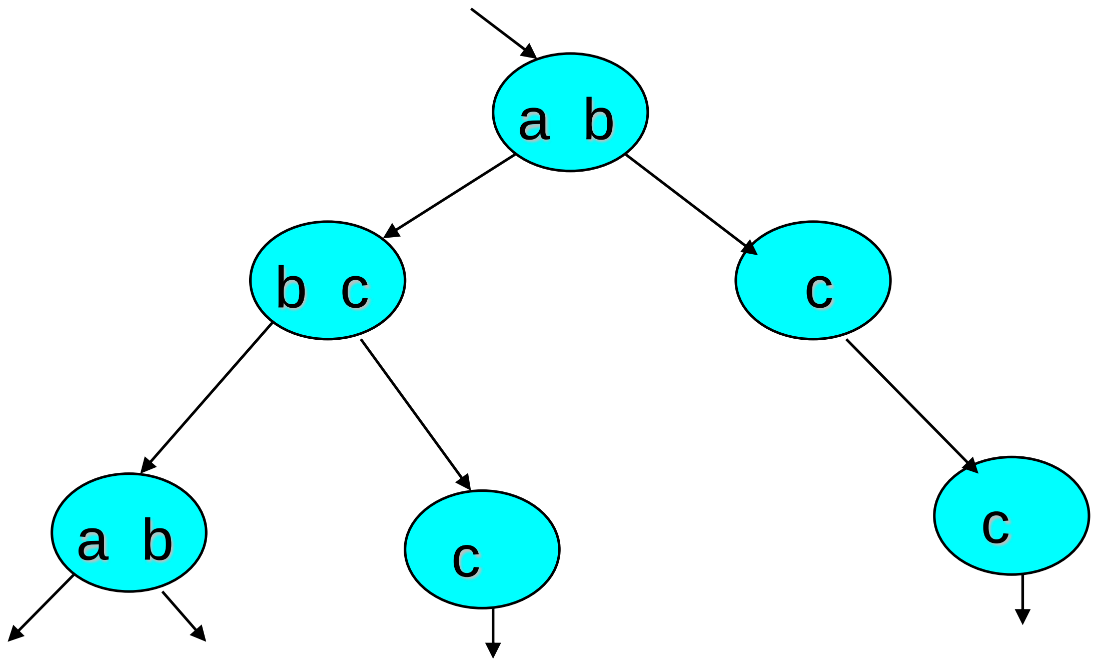
Branching Time Logic

Branching time paradigm:

- Interpreted over computation trees, not linear traces

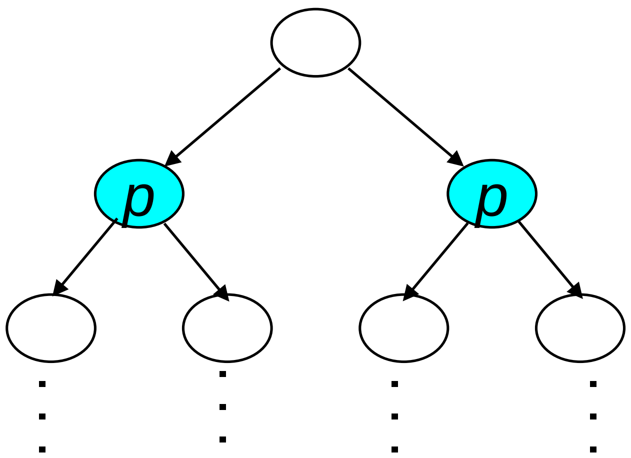
Computation tree:





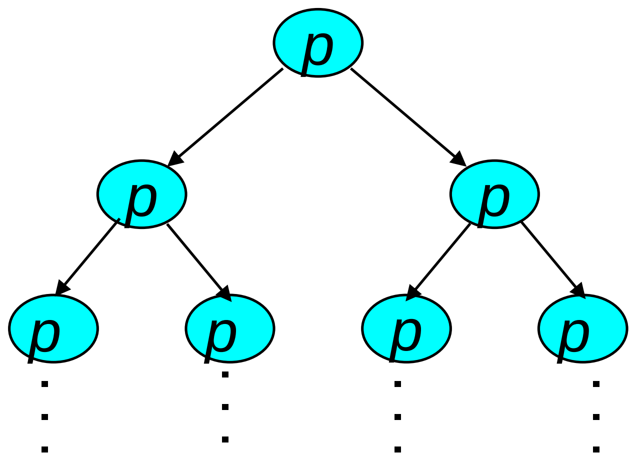
Universal Path Quantification

AX *p*



In all the next states *p* holds

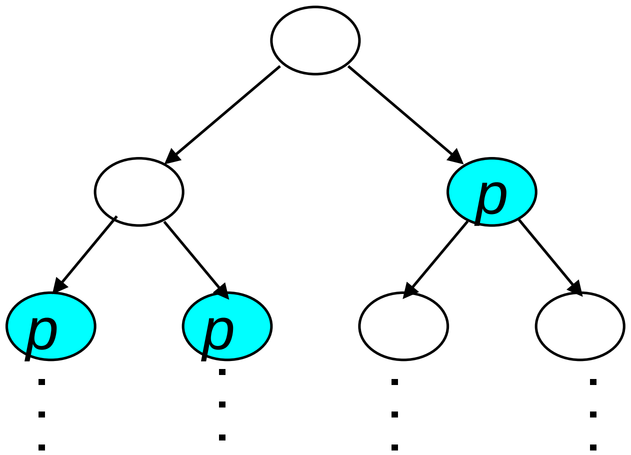
AG *p*



Along all the paths *p* holds forever

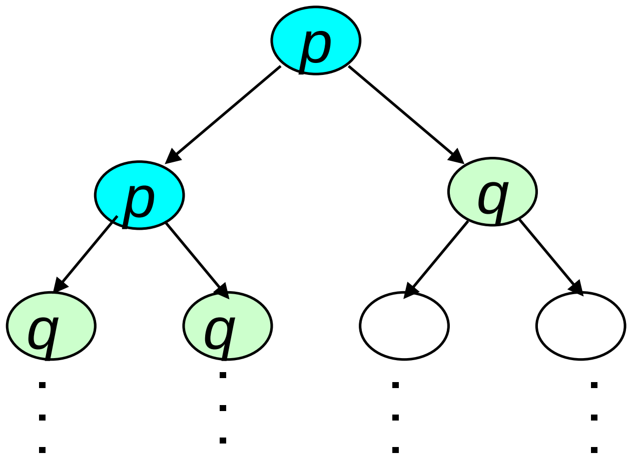
Universal Path Quantification

AF *p*



Along all the paths *p* holds eventually

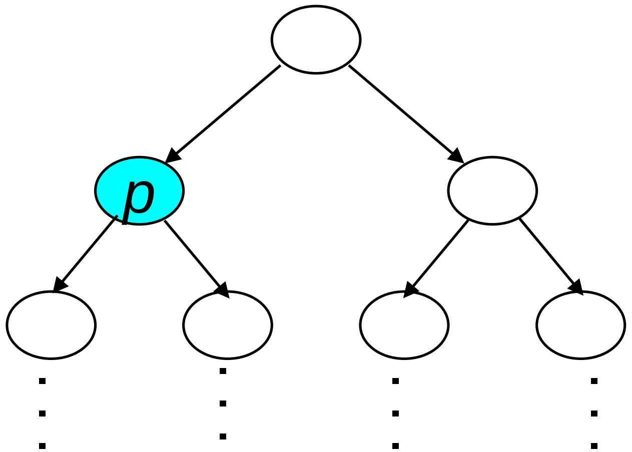
A(*p* U *q*)



Along all paths *p* holds until *q* holds

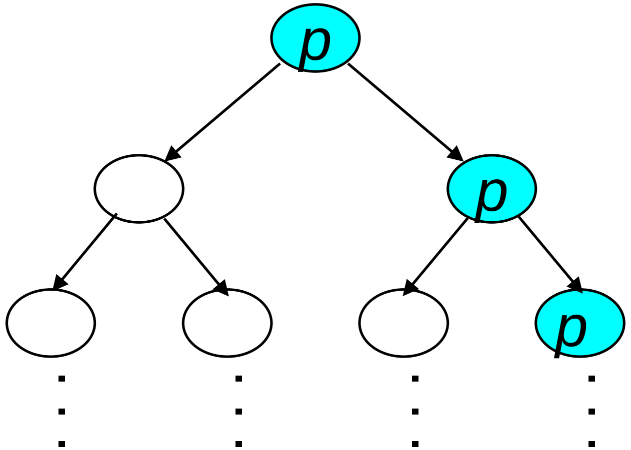
Existential Path Quantification

EX *p*



There exists a next state where *p* holds

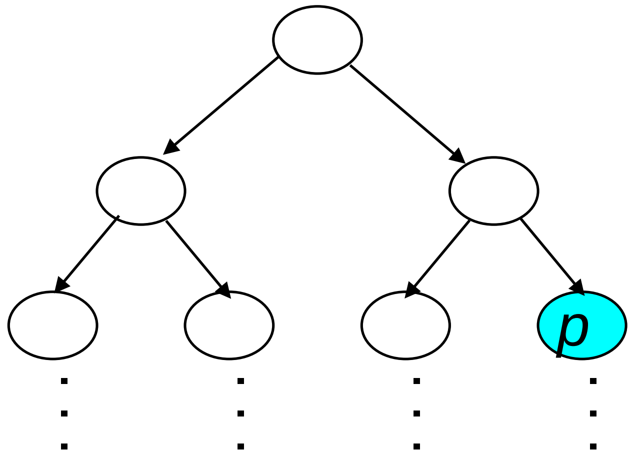
EG *p*



There exists a path along which *p* holds forever

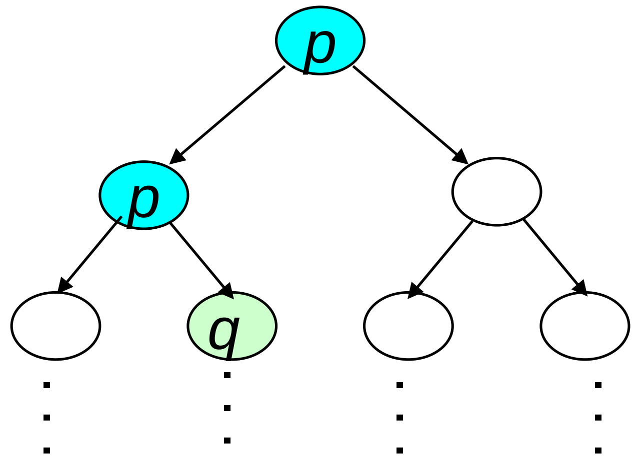
Existential Path Quantification

EF *p*



There exists a path along which *p* holds eventually

E(*p* U *q*)



There exists a path along which *p* holds until *q* holds

Computation Tree Logic (CTL)

Syntax:

- Given a set, AP, of atomic propositions:
  - All Boolean formulas over AP are CTL properties, and
  - If *f* and *g* are CTL properties, then so are ¬f, f∧g f∨g AXf, EXf, A[fUg] and E[fUg]
- We also have derived properties like EFg, AFg, EGf, and AGf

Semantics:

- The property Af is true at a state s of the Kripke structure, iff the path property f holds on all paths starting at s
- The property Ef is true at a state s of the Kripke structure, iff the path property f holds on some path starting at s

Nested Properties in CTL

AX AG p

“ from all the next states p holds forever along all paths ”

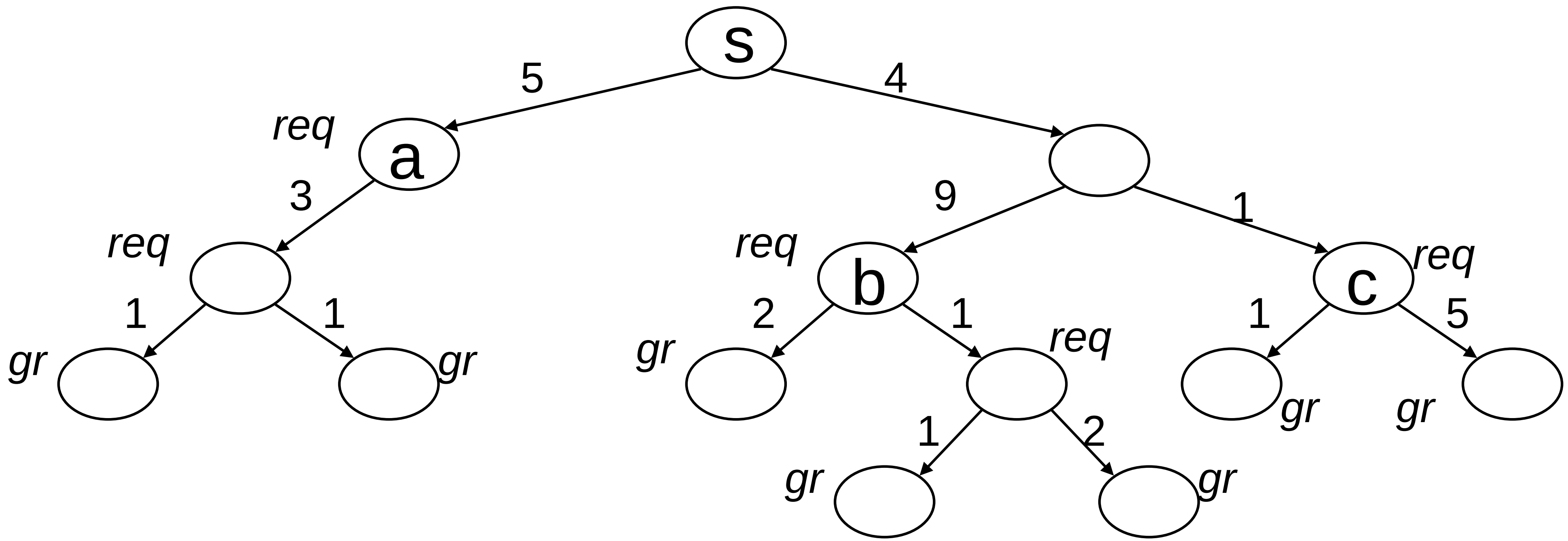
EX EF q

“ there exists a next state from which there exists a path to a state where q holds ”

AG EF r

“ from any state there exists a path to a state where r holds ”

Example: *Analyzing Request and Grants*



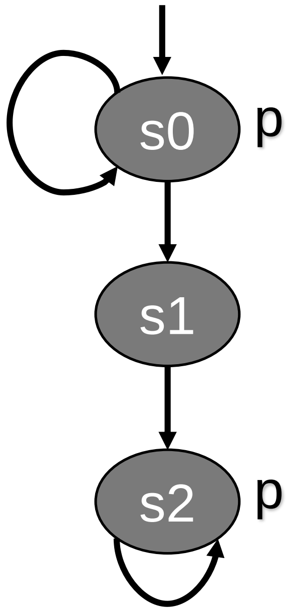
From s the system always makes a request in future:	$AF\ req$
All requests are eventually granted:	$AG(\ req \rightarrow AF\ gr\ )$
Sometimes requests are immediately granted:	$EF(\ req \rightarrow EX\ gr\ )$
Requests are not always immediately granted:	$\neg AG(\ req \rightarrow AX\ gr\ )$
Requests are held till grant is received:	$AG(\ req \rightarrow AF(\ req\ U\ gr\ )\ )$

LTL versus CTL

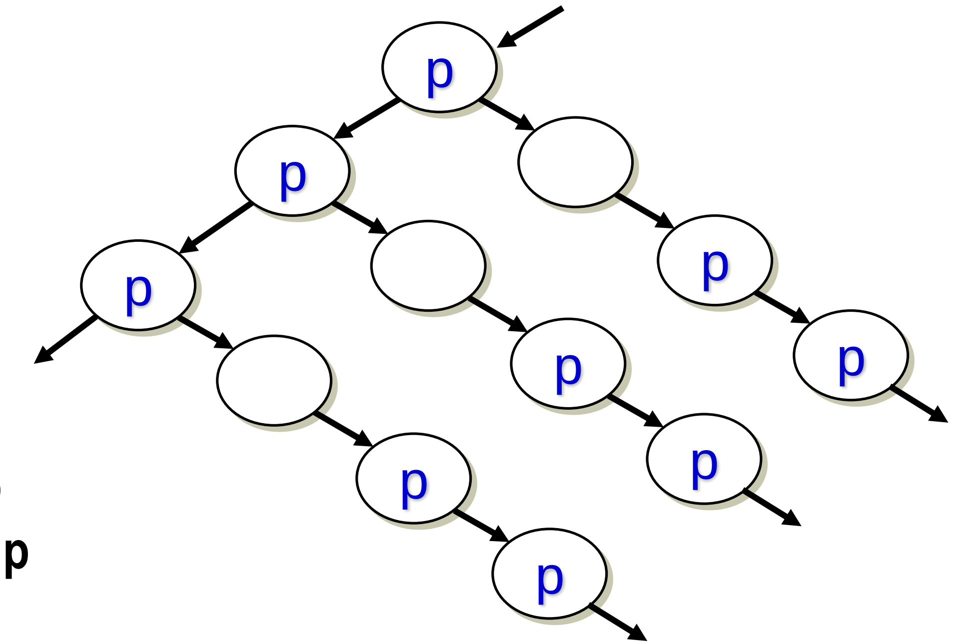
CTL has more operators than LTL – *which allows us to specify branching time properties (not supported in LTL).*

Can all LTL properties be expressed in CTL?

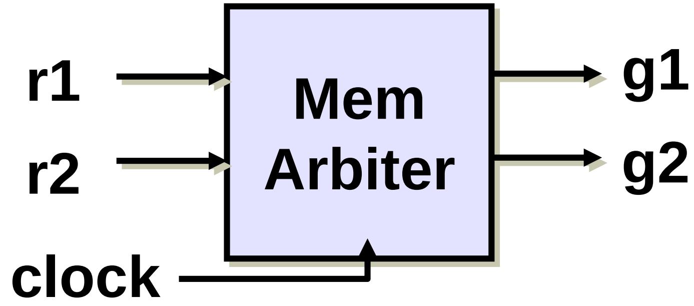
- **No.**
- **For example, FGp cannot be expressed in CTL**
- **Note that FGp is not equivalent to AFAGp**



Satisfies FGp but not AFAGp



Memory Arbiter: Specs



mem-arbiter(input r1, r2, clock, output g1, g2)

Properties:

1. Request line r1 has higher priority than request line r2. Whenever r1 goes high, the grant line g1 must be asserted for the next two cycles

   $G[\, r1 \Rightarrow Xg1 \wedge XXg1 \,]$

2. When none of the request lines are high, the arbiter parks the grant on g2 in the next cycle

   $G[\, \neg g1 \Rightarrow g2 \,]$

3. When r1 is low for consecutive cycles, then g1 should be low in the next cycle

   $G[\, \neg r1 \wedge X\neg r1 \Rightarrow XX \neg g1 \,]$

4. The grant lines g1 and g2 are mutually exclusive

   $G[\, \neg g1 \vee \neg g2 \,]$

SystemVerilog Assertions: A Quick Overview

```
property P1;
  @( posedge clk )
    r1 |→ ##1 g1 ##1 g1;
endproperty
```

```
property P2;
  @( posedge clk )
    !g1 |→ g2;
endproperty
```

LTL Properties:

P1: $G[r1 \Rightarrow Xg1 \wedge XXg1]$

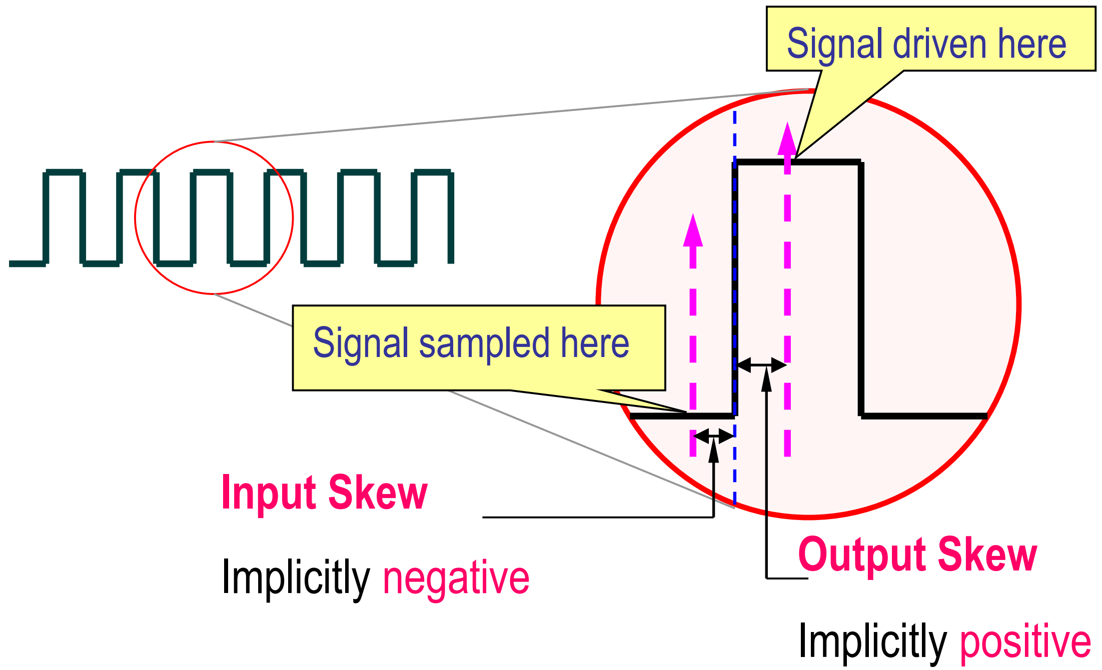
P2: $G[\neg g1 \Rightarrow g2]$

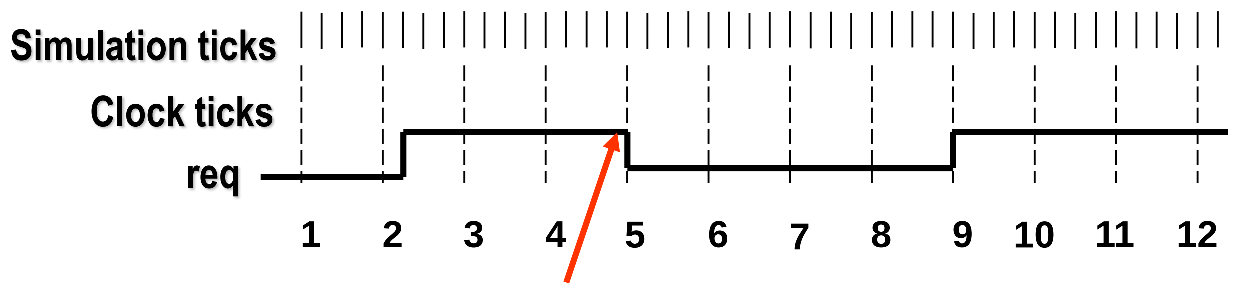
P3: $G[\neg r1 \wedge X \neg r1 \Rightarrow XX \neg g1]$

P4: $G[\neg g1 \vee \neg g2]$

```
property P3;
  @( posedge clk )
    !r1 ##1 !r1 |→ ##1 !g1;
endproperty
```

```
property P4;
  @( posedge clk )
    !g1 || !g2;
endproperty
```





1. Value of req at clock tick 5 is 1 not 0
2. Value of req at clock tick 9 is 0 not 1

SVA: Sequence Expressions

Sequence expressions are the basic building blocks of SVA

Examples:

```
##0 r1          // r1 is true in this cycle
##1 r1          // r1 is true in the next cycle
##5 r1          // r1 is true exactly after 5 cycles
##[5:9] r1      // r1 is true sometime between 5th and 9th cycle
```

Comparison with Timed LTL

- ##1 r1 is the same as $X\ r1$
- ##5 r1 is the same as $F_{[5,5]}\ r1$
- ##[5:9] r1 is the same as $F_{[5,9]}\ r1$

What is the meaning of the following sequence expression?

a ##[1:5] (b||c) ##3 d

Sequence expressions can be given a name

For example, we may rewrite a ##[1:5] (b||c) ##3 d as:

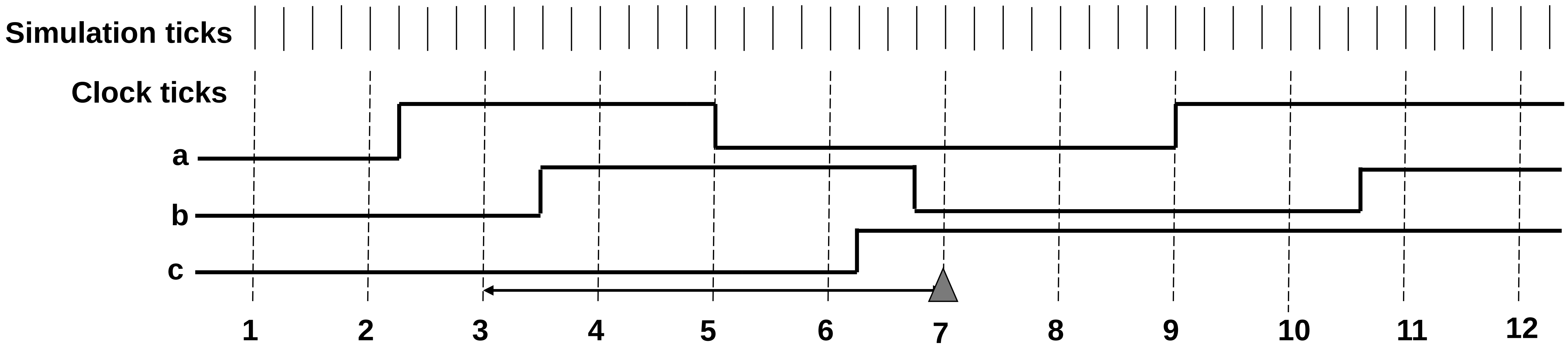
```
sequence s1;
  (b||c) ##3 d;
endsequence
sequence s2;
  a ##[1:5] s1;
endsequence
```

Note the use of s1 here

Sequence Operations: *Repetition*

Consecutive Repetition

- p[*5] — matches when 5 consecutive states satisfy p
- p[*3:5] ##1 q — k (3≤k≤5) consecutive matches followed by q
- p[*3:$] ##1 q — At least 3 consecutive matches followed by q
- *The request r must remain high until the grant g is asserted:* r |→ r[*1:$] ##1 g
- *The LTL property, p U q, is equivalent to:* p[*0:$] ##1 q ← Note the 0 here

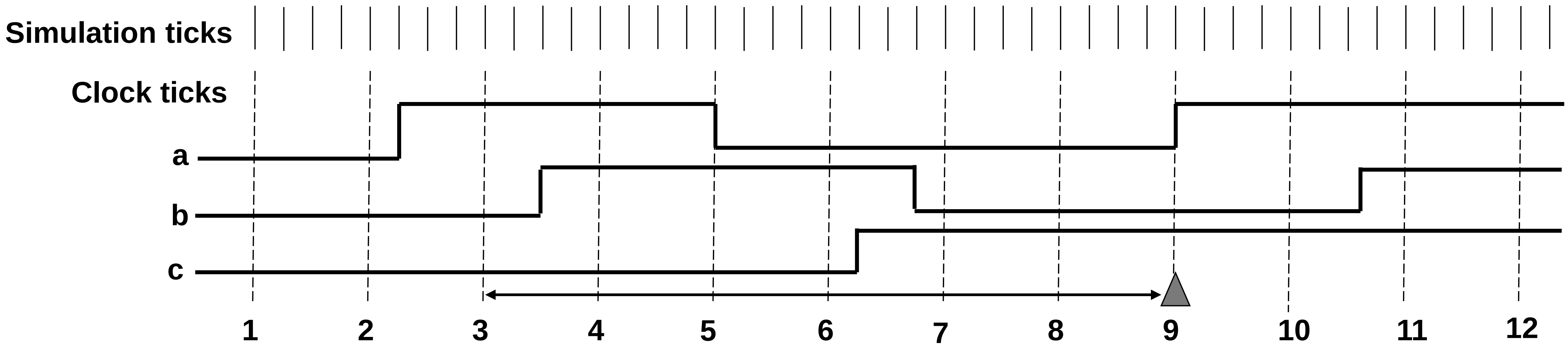


a ##1 b [*3] ##1 c

Sequence Operations: *Repetition*

Goto Repetition

- p[*→5] ##1 q   the match of q at some time t is preceded by 5 matches (not necessarily consecutive) of p, including one at time t – 1.
- ***The transfer must be aborted if the transfer is "split" more than once:*** split[*→2] ##1 abort
- p[*→3:5] ##1 q the match of q at some time t is preceded by 3 to 5 matches (not necessarily consecutive) of p, including one at time t – 1.

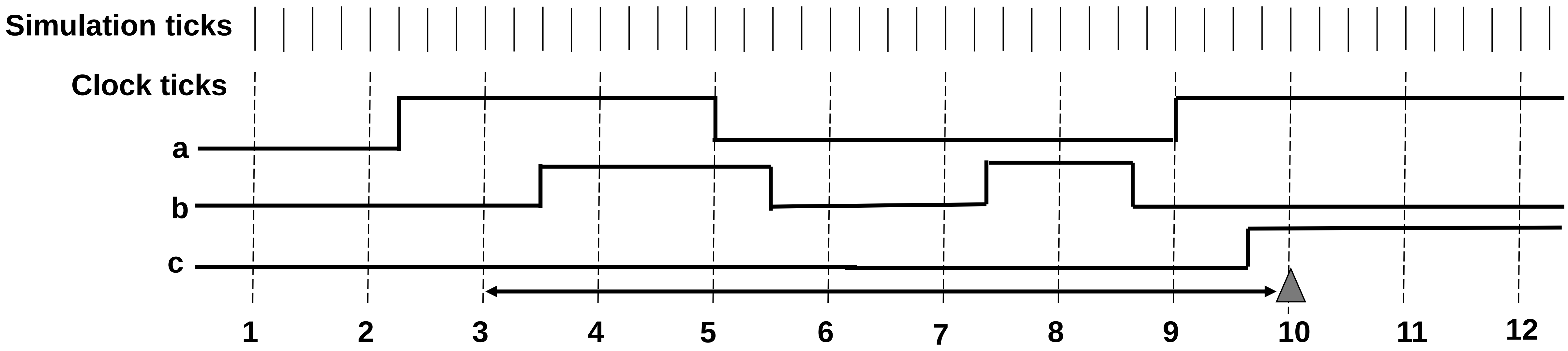


a ##1 b [*→3] ##1 c

Sequence Operations: *Repetition*

Non-consecutive Repetition

- split[*=2] ##1 abort — *The transfer is aborted if it is split more than once, but it is not necessary that the abort takes place immediately after the second split.*
- p[*=3:5] ##1 q — matches at time t, if q matches at time t and p matches 3 to 5 times before time t.



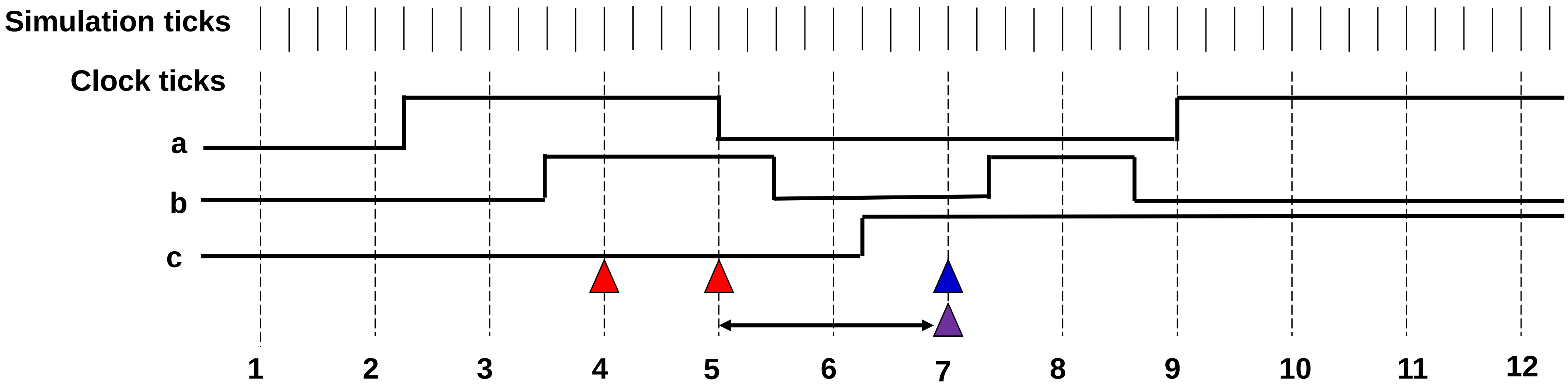
a ##1 b [*=3] ##1 c

AND – operation

- The binary operator and is used when both operand expressions are expected to succeed
- End time of the operands can be different

Example:

(a ##1 b) and (a ##1 b ##2 c)

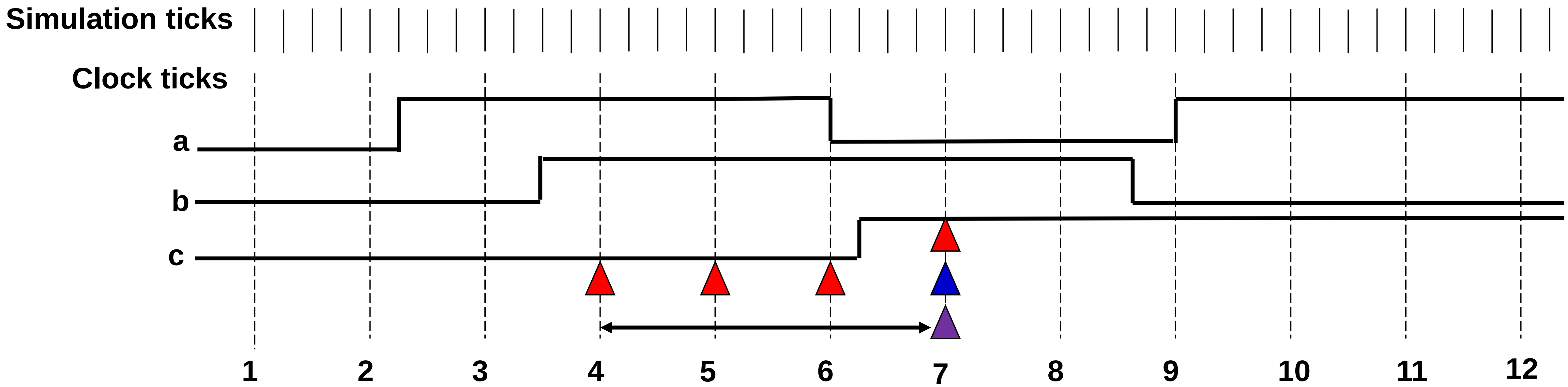


Intersection – operation

- The binary operator intersect is used when both operand expressions are expected to succeed
- End times of the operand expressions must be the same
- Length of the two operand sequences must be same

Example:

(a ##1 b) intersect (a ##1 b ##2 c)

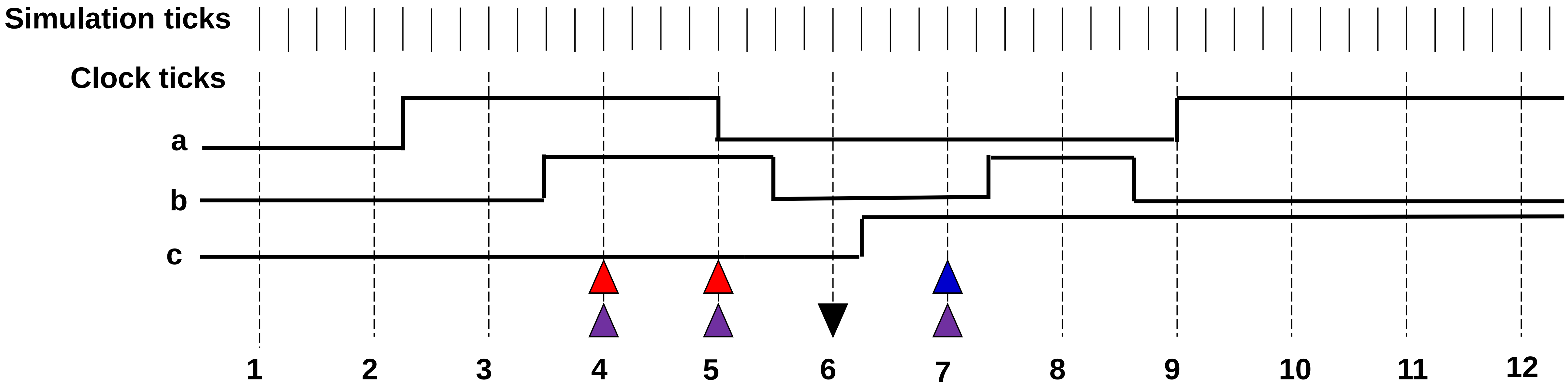


OR – operation

- The binary operator or is used when at least one of operand expressions are expected to match
- End timed of the operand can be different

Example:

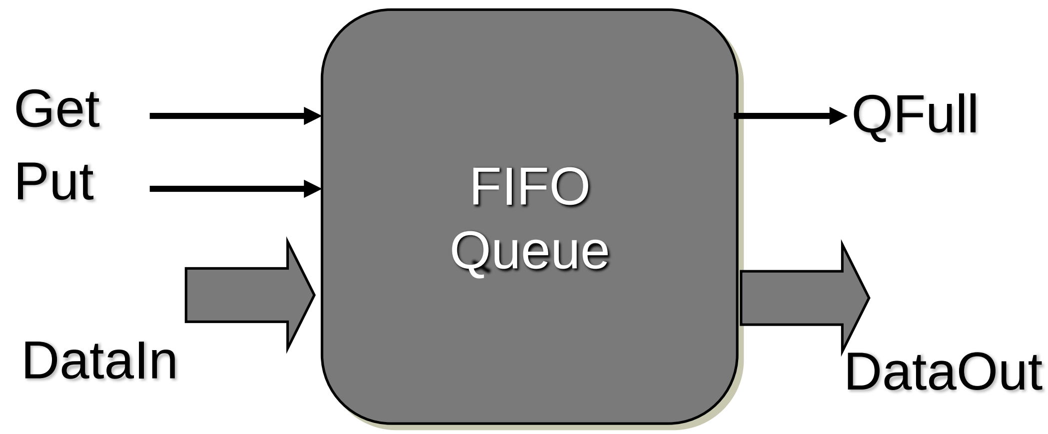
(a ##1 b) or (a ##1 b ##2 c)



Local Variables

Property*:*

If X and Y are any two data items such that X was pushed before Y, then X will come out of the queue before Y




```
property FIFO_check;
   int x;
   int y;
   @( posedge clk )
     (( Put && !QFull, x = DataIn ) ##[1,$] ( Put && !QFull, y = DataIn )) |→
         ##[1,$] (( Get && x == DataOut ) ##[1,$] (Get && y == DataOut )) ;
endproperty
```

Few More Constructs in SVA

Two types of implications

- **Overlapped Implication Operator:**

  In the property, s1 |→ s2, the match of s2 starts from the same cycle as the one in which we complete a match for s1.

- **Non-overlapped Implication Operator:**

  In the property, s1 |=> s2, the match of s2 starts from the cycle *after* the one in which we complete a match for s1.

Use of *disable-iff*

y must be asserted within 16 cycles of x, unless reset is asserted in between

```
property DisableOnReset;
    @(posedge clk)    disable iff (reset)   x |→ ##[1:16] y;
endproperty
```

Immediate and Concurrent Assertions

Immediate Assertions

- Immediate assertions follow simulation event semantics for their execution
- Immediate assertions are executed like a statement in a procedural block

```
assert (expression) Action_block
Action_block ::= statement_or_null | [statement] else statement
```

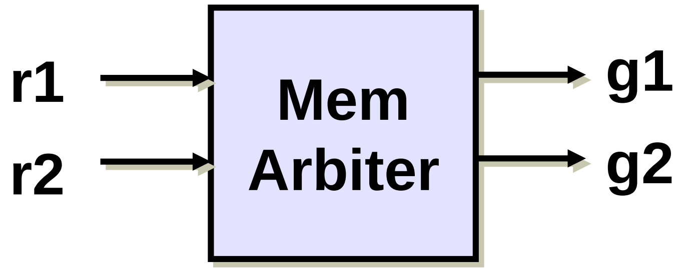
Concurrent Assertions

- Describe behavior that spans over time
- Evaluation model is based on a clock
- The values of variables used are the sampled values in the specified clock edge

```
prop_p1: assert property (p1) pass_stat else fail_stat
```

Assert (guarantee) and *Assume (constraint)* Properties

Example: *Every low priority request, r2, is eventually granted by the arbiter*




```
property NoStarvation;
   @(posedge clk)   r2 |→ ##[1:$] g2 ;
end property
AssertNoStarvation: assert property (NoStarvation);
```

This requirement conflicts with our earlier property P1:

```
property P1;
   @(posedge clk)   r1 |→ ##1 g1 ##1 g1;
endproperty
GrantWhenRequest: assert property (P1);
```

- If any *assume* property fails, then monitoring of the *assert* properties become redundant
- *assume* properties may be used to prune the state space before checking the *assert* properties in formal verification

Suppose we are now given with assumption that whenever g1 is asserted, r1 remains low for the next 4 cycles

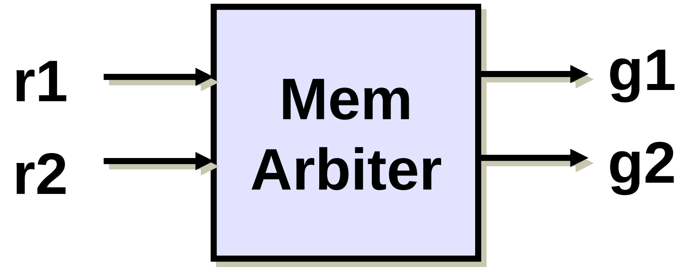
```
property FairnessOfr1;
   @(posedge clk)   g1 |→(!r1) [*4] ;
endproperty
AssumeR1IsFair: assume property (FairnessOfr1);
```

Under assumption AssumeR1IsFair, there is no conflict between the properties GrantWhenRequest and AssertNoStarvation

Cover Properties – Coverage Specifications in SVA

- The property P4 is interpreted non-vacuously only when r1 is low in two consecutive cycles (Vacuity rules are applied only to the implication operator)

```
property P4;
  @(posedge clk)   !r1 ##1 !r1 |→ ##1 !g1;
endproperty
coverP4: cover property (P4);
```



- Coverage Results contain:
  - Number of times attempted
  - Number of times succeeded
  - Number of times failed
  - Number of times succeeded for vacuity
  - Each attempt with an attemptID and time
  - Each success/failure with an attemptID and time

Only for Implication Properties

Multiple Clock Support in SVA

Multiple clock is allowed in

- Concatenation of two sequences, where each sequence can have a different clock

```
sequence s1;
        @(posedge clk0) sig0   ##   @(posedge clk1) sig1;
endsequence
```

- The antecedent of an implication on one clock, while the consequent is on another clock

```
property s2;
        @(posedge clk0) sig0 |=>   @(posedge clk1) sig1;
endproperty
```

Architectural Styles for Assertion IPs

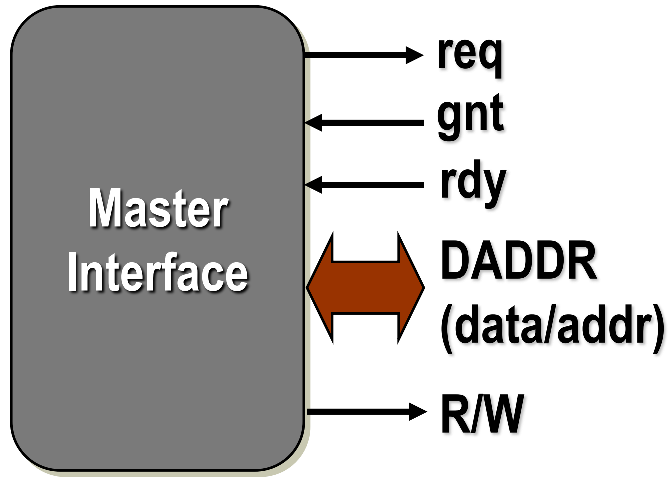
Event-based Specifications

- Only properties defined over interface signals

State-based Specifications

- Auxiliary state machines (ASM)
- Properties specified using state-bits of ASM and interface signals

The MyBus Protocol



Address and data multiplexed

Master asserts req, waits for gnt

Address Cycle: Then it floats the address and waits for rdy from slave

Data Cycle: On receiving rdy, it expects data in next cycle (if READ), or floats data in next cycle (if WRITE)

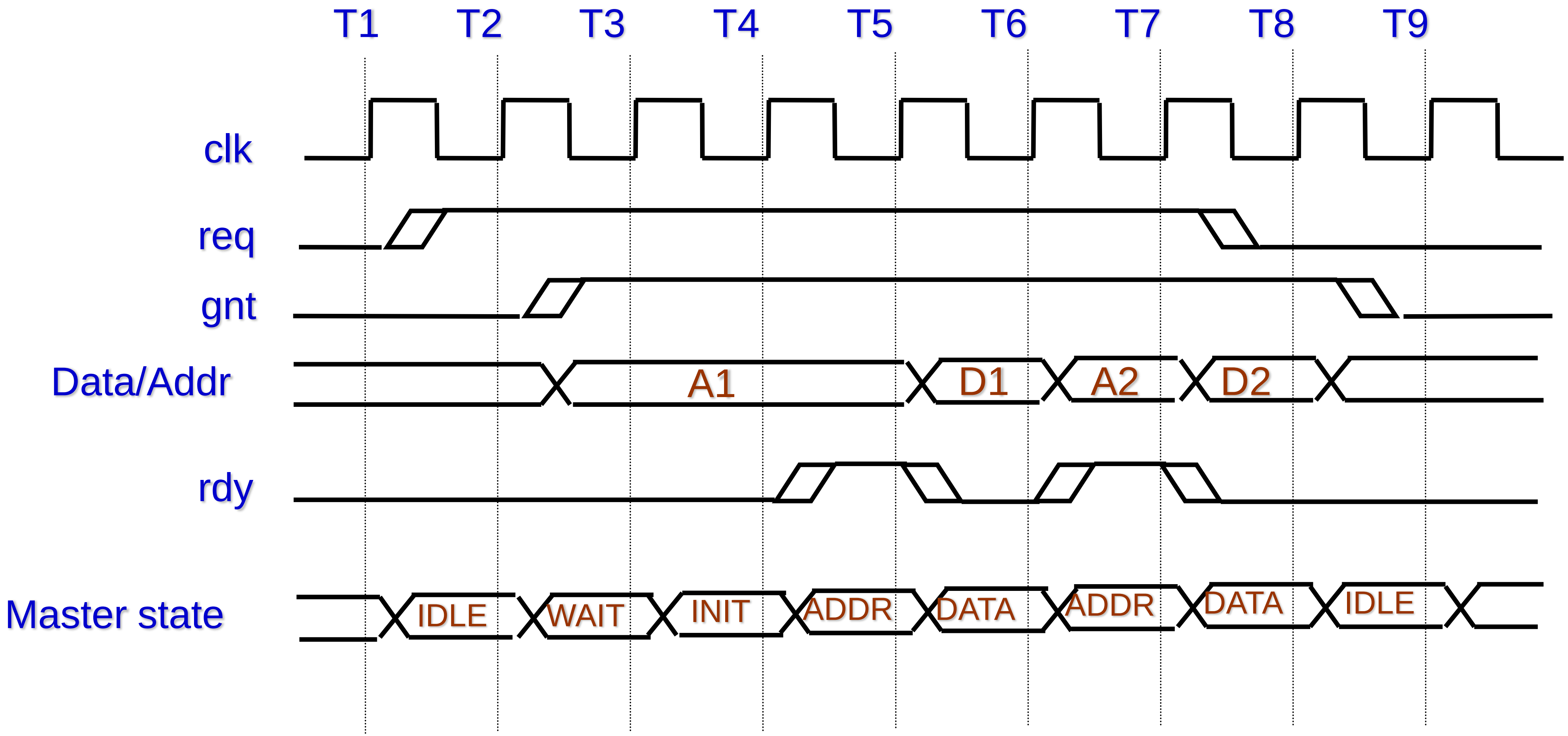
R/W indicates intent: read/write

After each data cycle, the master may start another address cycle by floating the next address

Properties:

- The protocol is non-preemptive. Once granted, the master owns the Bus until it lowers its *req* line
- If the master is in the ADDRESS cycle, it should not change the address floated in the Bus until it receives the *rdy* signal from the slave
- Each DATA cycle is of unit cycle duration

A simple Bus Transfer



Event-based Coding

The protocol is non-preemptive. Once granted, the master owns the Bus until it lowers its *req* line

```
property NoPreemption;
        @(posedge clk)   $rose(gnt) |→ ##1 gnt [*1:$] ##0 !req ;
endproperty
```

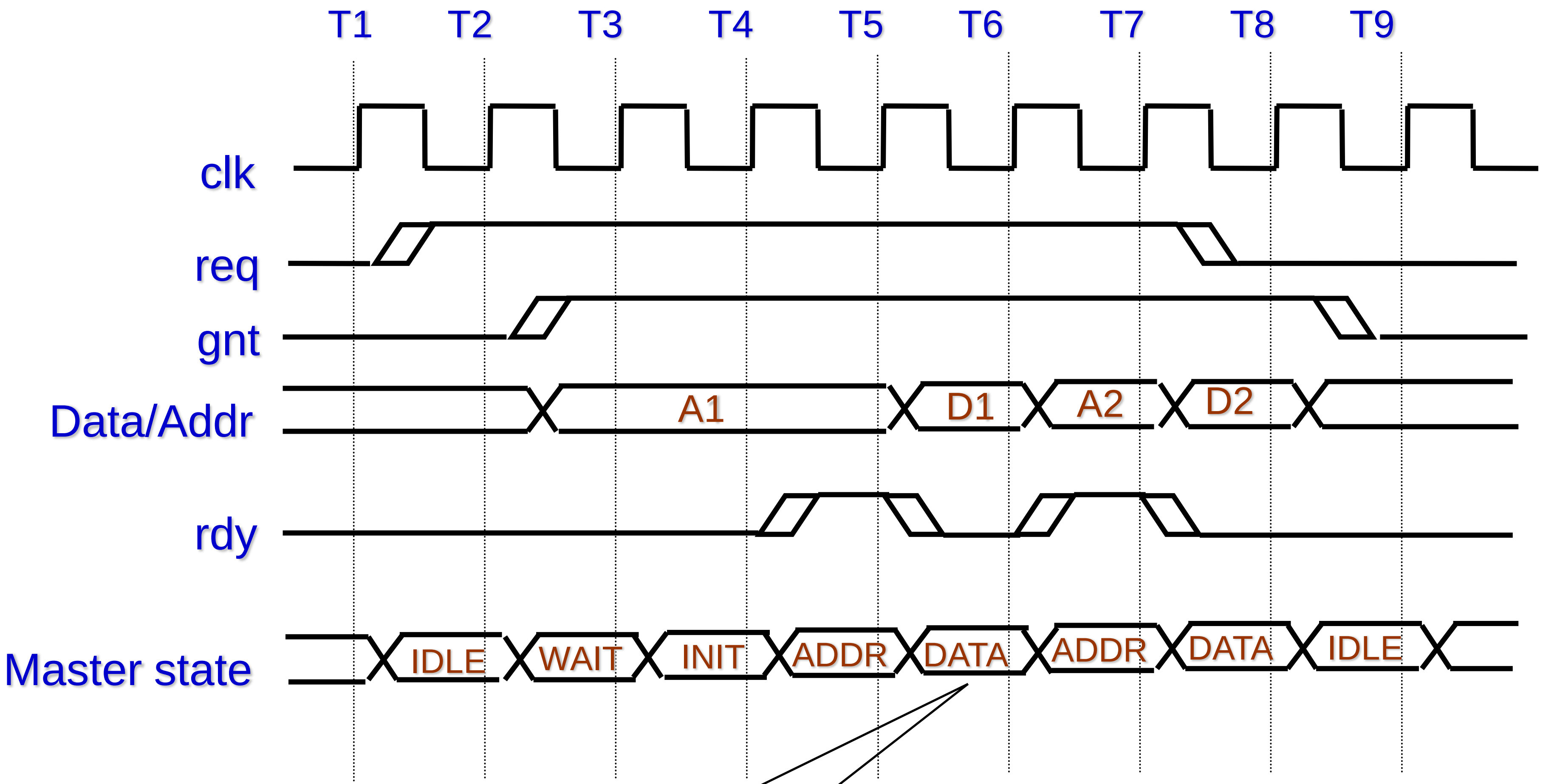
$rose(gnt) is true in a cycle if the signal *gnt* rose in that cycle

If the master is in the ADDRESS cycle, it should not change the address floated in the Bus until it receives the *rdy* signal from the slave

```
property IncorrectAddressStable;
        int x;
        @(posedge clk)   (req && gnt && !rdy, x = DADDR) |→ ##1 (x == DADDR) ;
endproperty
```

This coding is not correct, since (req && gnt && !rdy) may be true at other places also.

The Problem with Event-based Coding




```
property IncorrectAddressStable;
  int x;
  @(posedge clk) (req && gnt && !rdy, x = DADDR) |→ ##1 (x == DADDR) ;
endproperty
```

The Context is Important …

What's the problem with this property?

```
property IncorrectAddressStable;
  int x;
  @(posedge clk) (req && gnt && !rdy, x = DADDR) |→ ##1 (x == DADDR) ;
endproperty
```

- We want to check this property only in the ADDRESS cycles, not in the DATA cycles
- How should be distinguish between an ADDRESS cycle and a data cycle?

```
property AddressStable;
  int x;
  @(posedge clk) (req && gnt && !rdy && !$fell(rdy), x = DADDR) |→ ##1 (x == DADDR) ;
endproperty
```

Demerits of Event-based Coding → State-based Coding

Each DATA cycle is of unit cycle duration

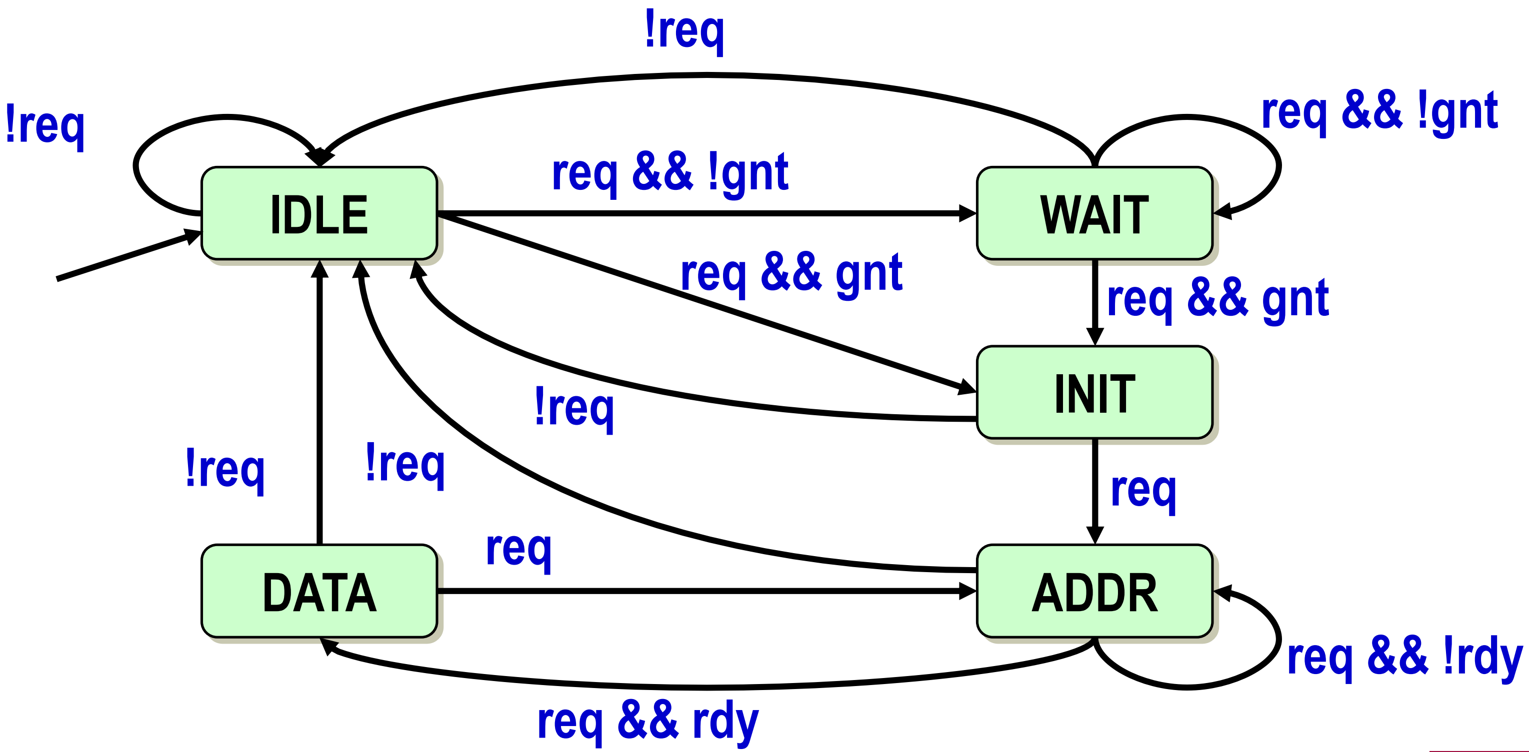
```
property SingleCycleDataTransfer;
   @(posedge clk)   (gnt && $fell(rdy)) |→ ##1 (!gnt || !$fell(rdy)) ;
endproperty
```

- The expression (gnt && $fell(rdy)) characterizes a DATA cycle. *Not obvious*!!

State-based Coding:

- Characterizing the context is a major problem in event-based coding
- In state-based coding we use an auxiliary state machine to capture the contexts and the transitions between them
  - We use the state labels for coding the actual properties
  - Improves readability and also Reduces coding errors

Auxiliary State Machine and State-based Coding



ASM

```
property SingleCycleDataTransfer;
  @(posedge clk)
  (state == 'DATA) |→ ##1 !(state == 'DATA) ;
endproperty
```

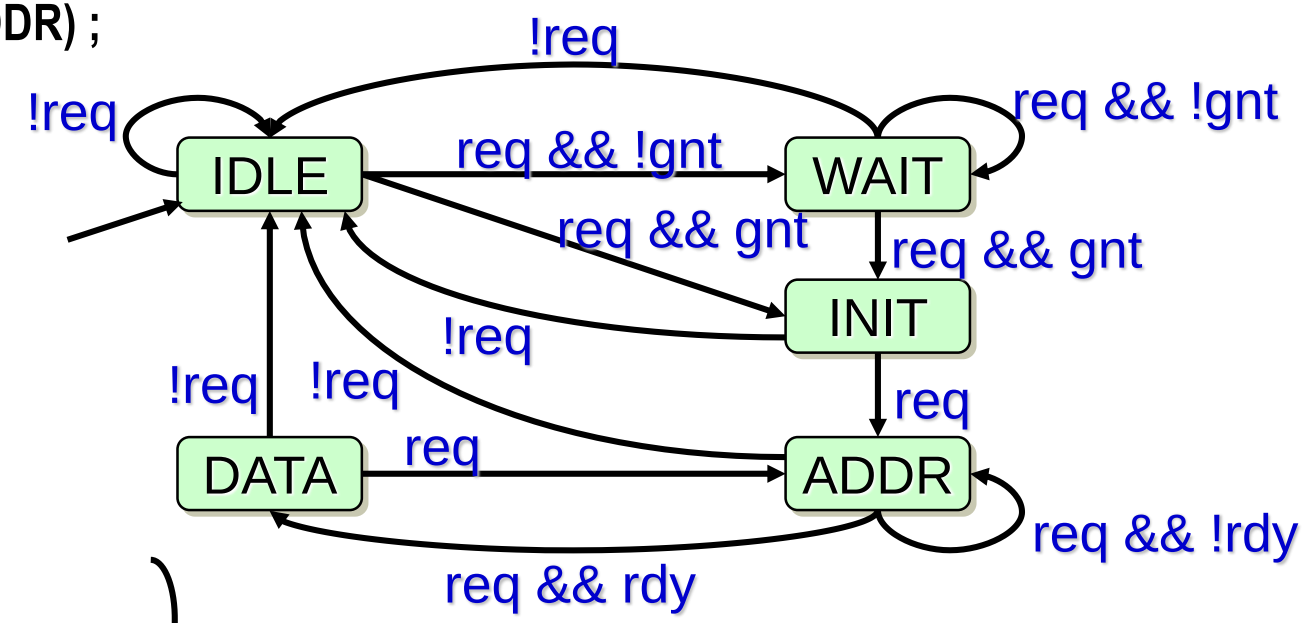
```
property AddressStable;
  int x;
  @(posedge clk)
  (state == 'ADDR, x = DADDR)
              |→ ##1 (x == DADDR) ;
endproperty
```

Encoding the Auxiliary State Machine

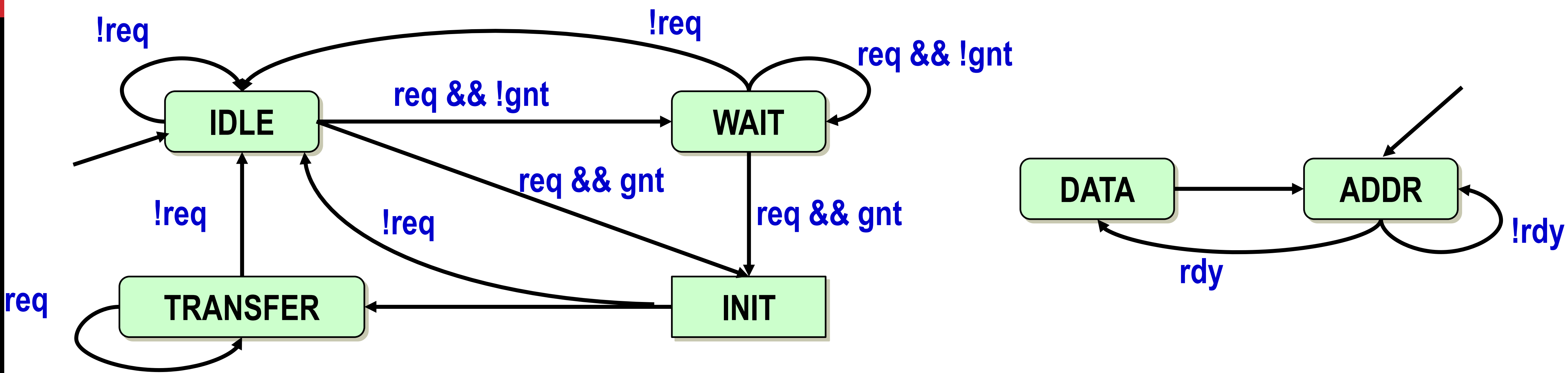
```
interface MasterInterface( input req, gnt, rdy, clk, int DADDR) ;
logic [2:0] state;
'define IDLE 3'b000
'define WAIT 3'b001
'define INIT 3'b010
'define ADDR 3'b011
'define DATA 3'b100
always @( posedge clk )
  case (state)
        'IDLE: state <= req? (gnt? 'INIT : 'WAIT) : 'IDLE;
        'WAIT: state <= req? (gnt? 'INIT : 'WAIT) : 'IDLE;
        'INIT: state <= req? 'ADDR : 'IDLE;
        'ADDR: state <= req? (rdy? 'DATA : ' ADDR) : 'IDLE;
        'DATA: state <= req? 'ADDR : 'IDLE;
  endcase
initial begin state = 'IDLE; end
endinterface
```

State encoding

State transition relation



Factored State Machines




```
property AddressStable;
  int x;
  @(posedge clk) (state1 == 'TRANSFER && state2 == 'ADDR, x = DADDR)
                 |-> ##1 (x == DADDR) ;
endproperty
```

```
property SingleCycleDataTransfer;
  @(posedge clk)
  (state1 == 'TRANSFER && state2 == 'DATA) |-> ##1 !(state2 == 'DATA) ;
endproperty
```

Regular expressions

- Let Σ be an alphabet with $\mathbf{A} \in \Sigma$

- Regular expressions over Σ have ***syntax:***

$$E ::= \underline{\phi} \mid \underline{\varepsilon} \mid \underline{A} \mid E + E' \mid E \,.\, E' \mid E^*$$

- The **semantics** of regular expression $\boldsymbol{E}$ is a language $\boldsymbol{L(E)} \subseteq \boldsymbol{\Sigma^*}$:

$$\boldsymbol{L(\underline{\phi}) = \phi^*} \qquad \boldsymbol{L(\underline{\varepsilon}) = \{\varepsilon\}} \qquad \boldsymbol{L(\underline{A}) = \{A\}}$$

$$\boldsymbol{L(E + E') = L(E) \cup L(E')} \qquad \boldsymbol{L(E.E') = L(E).L(E')} \qquad \boldsymbol{L(E^*) = L(E)^*}$$

Syntax of ω-regular expressions

- *Regular expressions* denote languages of finite words
- *ω-Regular expressions* denote languages of infinite words
- An *ω-regular expression* G over Σ has the form:

$$G = E_1 . F_1^{\omega} + \ldots + E_n . F_n^{\omega} \quad \text{for} \;\; n>0$$

  - where E_i, Fi are regular expressions over Σ with $\varepsilon \notin L(F_i)$

- Some examples:
  - $(A + B)^* . B^{\omega}$,
  - $(B^* . A)^{\omega}$,
  - $A^* . B^{\omega} + A^{\omega}$

Semantics of ω-regular expressions

- ***For*** $L \subseteq \Sigma^*$ ***let*** $L^\omega = \{w_1 w_2 w_3 \ldots \mid \forall i \geq 0.\, wi \in L\}$
- ***Let ω-regular expression*** $G = E_1.F_1^\omega + \ldots + E_n.F_n^\omega$
- **The *semantics* of *G* is the language** $L_\omega(G) \subseteq \Sigma^\omega$:

$$L_\omega(G) = L(E_1).L(F_1)^\omega \cup \ldots \cup L(E_n).L(F_n)^\omega$$

- **G_1 and G_2 are *equivalent*, denoted** $G_1 \equiv G_2$ **, if** $L_\omega(G_1) = L_\omega(G_2)$

ω-Regular languages

- L is ω-regular if $L = L_\omega(G)$ for some ω-regular expression G
- Examples over $\Sigma = \{A, B\}$:
  - Language of all words with infinitely many As: $(B^*.A)^\omega$
  - Language of all words with finitely many As: $(A+B)^*.B^\omega$
  - The empty language: $\emptyset^\omega$
- ω-Regular languages are closed under ∪ , ∩ and complementation

ω-Regular safety properties

- **Definition:**

LT property P over AP is ω-Regular if
P is an ω-regular language over the alphabet 2^{AP}

- **Or, equivalently:**

LT property P over AP is ω-Regular if
P is a language accepted by a nondeterministic Büchi automaton over 2^{AP}