DOULOS KnowHow WEBINARS

Delivering KnowHow www.doulos.com

How nondeterminism accelerates formal verification

Presenter: Doug Smith
Engineer / Instructor

Webinar partner:

How nondeterminism accelerates formal verification

- ➡ Randomness in formal
- Nondeterminism accelerates formal
- Wrap-up

DOULOS

A Case for Randomness

- Tests what we don’t know
- Faster test development
- Find corner cases faster
- Deeper state space exploration

Not Just for Simulation

Determinism in algorithms is reproducibility of results with the same inputs

Simulation

- Order of process execution is nondeterministic
- Everything else is deterministic - random stability + reproducibility

Formal

- Algorithms are nondeterministic – different results every time!
- I.e., *randomness* is built into formal by default

Randomness is not just for simulation!

Some Things Aren't So Different

Randomization in …

Simulation	Formal
Write random constraints	Constrain formal's randomness
Test what we don't know	Frees formal to test everything
Hit corner cases faster	Skips straight to target
Coverage shows what was hit	Coverage shows formal's path

But Wait, There's More

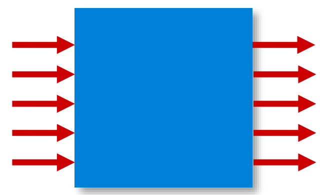
Randomness (nondeterminism) in formal also helps with …

- Reducing the cone of influence
- Abstracting away complexity
- Simplifying property writing
- Dealing with inconclusives

Nondeterminism in Formal

Anything undriven is a formal control point (random)

DUT inputs



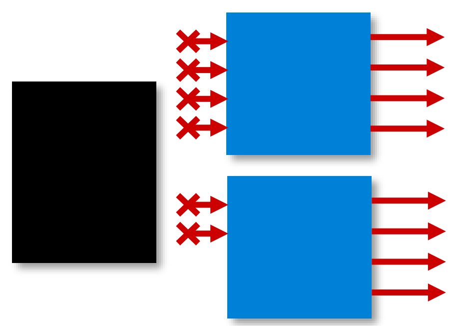
Cut signals

```
stopat counter
```

```
snip_driver counter
```

```
netlist cutpoint counter
```

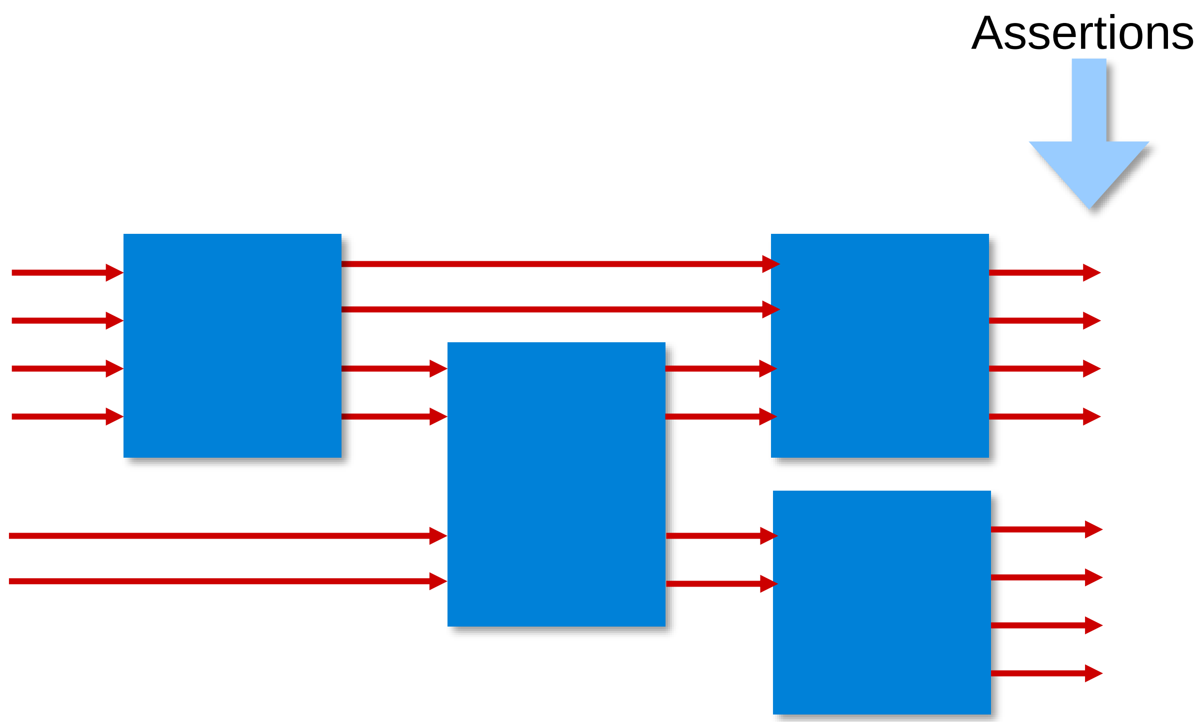
Black-boxed ports



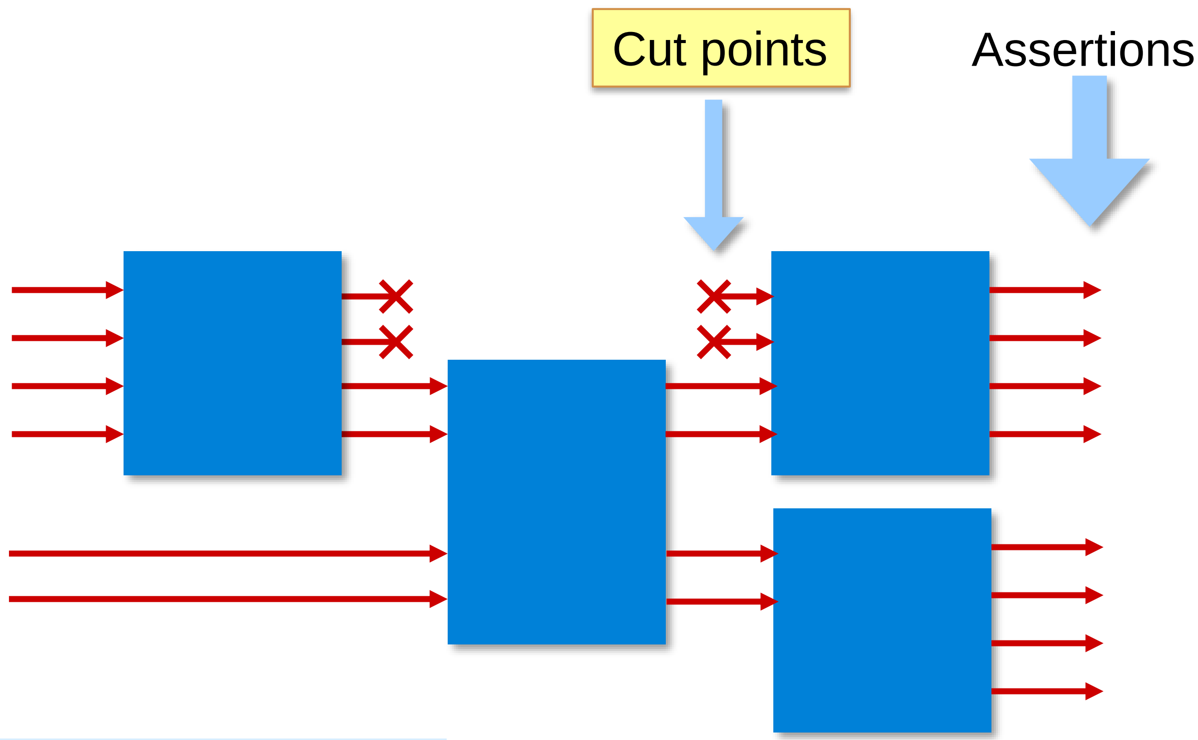
Undriven signals and variables

```
logic [WIDTH-1:0] counter;
```

Cut Points and Black Boxes



Cut Points and Black Boxes

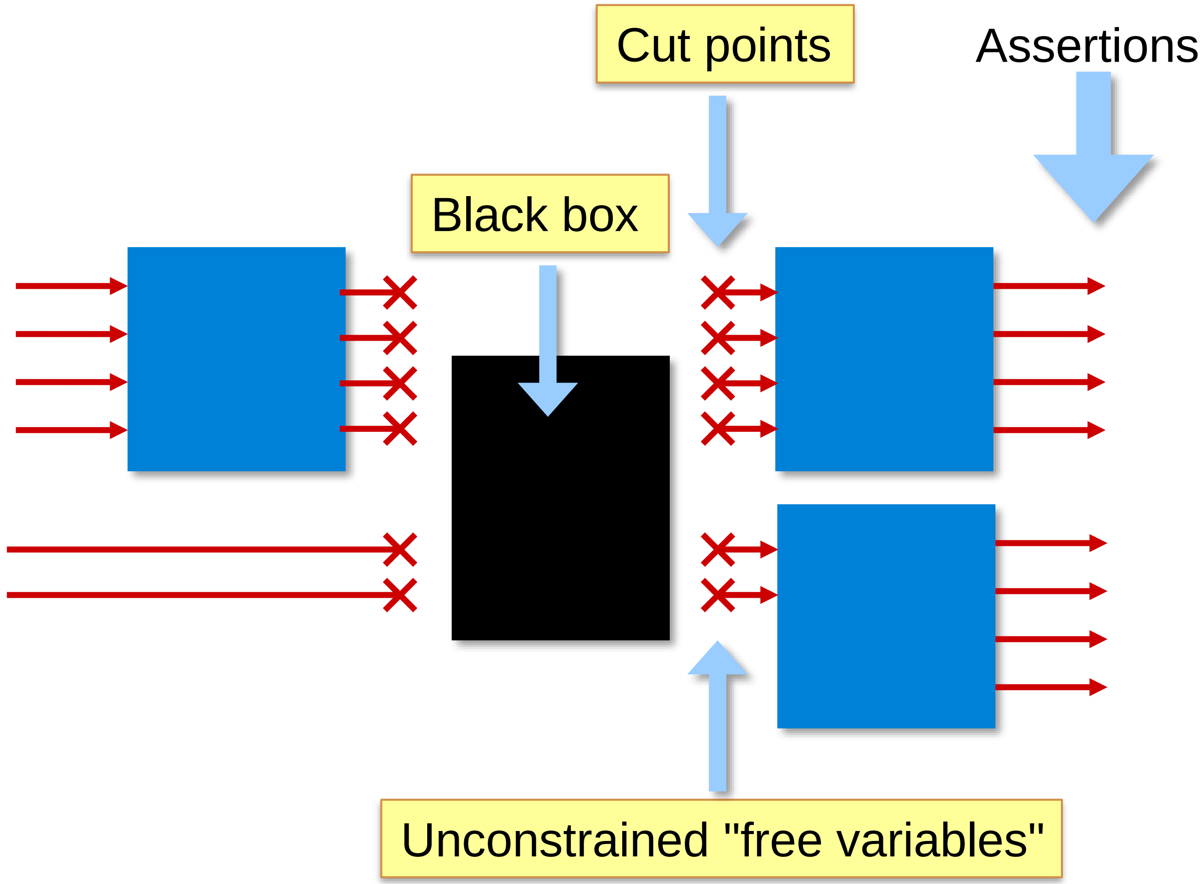



```
stopat signal
```

```
snip_driver signal
```

```
netlist cutpoint signal
```

Cut Points and Black Boxes



Initial Value Abstraction

```
always @(posedge clock or posedge reset)
  if (reset)
    count <= 0;
  else if (count < 16'hffff)
    count <= count + 1'b1;
  else
    count <= 0;

assign en1 = (count == 16'h01ff);
assign en2 = (count == 16'h07ff);
assign en3 = (count == 16'h3fff);

always @(posedge clock)
begin
  if (en1)
    array[address1] <= data_in;    // Bounds check
  if (en2)
    array[address2] <= data_in;    // Bounds check
  if (en3)
    array[address3] <= data_in;    // Bounds check
end
```

With no reset, count is a free variable

```
reset -none
```

```
reset none
```

```
formal init {}
```

Fails at depth=1

Fails at depth=1

Fails at depth=1

Free Variables

Known as *random, free, or nondeterminism (ND)* variables

Undriven signals and variables are formal control points

```
module formal_tb( ... [7:0] output_data );
   byte data;                      // Data output
   byte random_data;               // Undriven variable => random

   always @(posedge clk or negedge rst_n)
      if ( !rst_n )
         data <= '0;
      else begin
         data <= random_data;
```

Formal picks a random value

```
assume property ( output_data == data );
```

Constrain output

How nondeterminism accelerates formal verification

- Randomness in formal
- ➡ Nondeterminism accelerates formal
- Wrap-up

DOULOS

How nondeterminism accelerates formal verification

Nondeterminism accelerates formal

- Reducing the cone of influence
- Simultaneous testing
- Faster property development
- Handling inconclusives
- Abstractions

DOULOS

Design Symmetry

```
module arbiter #(...) ...

  always @* begin
     case ( pointer_reg )
       2'b00 :
              if (req[0]) grant = 4'b0001;
         else if (req[1]) grant = 4'b0010;
         else if (req[2]) grant = 4'b0100;
         else if (req[3]) grant = 4'b1000;
         else             grant = 4'b0000;
       2'b01 :
              if (req[1]) grant = 4'b0010;
         else if (req[2]) grant = 4'b0100;
         else if (req[3]) grant = 4'b1000;
         else if (req[0]) grant = 4'b0001;
         else             grant = 4'b0000;
       2'b10 :
              if (req[2]) grant = 4'b0100;
         else if (req[3]) grant = 4'b1000;
         else if (req[0]) grant = 4'b0001;
         else if (req[1]) grant = 4'b0010;
         else             grant = 4'b0000;
       2'b11 :
              if (req[3]) grant = 4'b1000;
         else if (req[0]) grant = 4'b0001;
         else if (req[1]) grant = 4'b0010;
         else if (req[2]) grant = 4'b0100;
         else             grant = 4'b0000;
     endcase
  end
```

Code repetition

Parameterization may repeat code

```
logic [1:0] pointer_reg;
always @(posedge clk or posedge reset)
   if ( reset )
      pointer_reg <= 0;
   else
      pointer_reg <= pointer_reg + 2'b1;
```

Data Independence

```
module arbiter #(...) ...

  always @* begin
     case ( pointer_reg )
       2'b00 :
              if (req[0]) grant = 4'b0001;
         else if (req[1]) grant = 4'b0010;
         else if (req[2]) grant = 4'b0100;
         else if (req[3]) grant = 4'b1000;
         else             grant = 4'b0000;
       2'b01 :
              if (req[1]) grant = 4'b0010;
         else if (req[2]) grant = 4'b0100;
         else if (req[3]) grant = 4'b1000;
         else if (req[0]) grant = 4'b0001;
         else             grant = 4'b0000;
       2'b10 :
              if (req[2]) grant = 4'b0100;
         else if (req[3]) grant = 4'b1000;
         else if (req[0]) grant = 4'b0001;
         else if (req[1]) grant = 4'b0010;
         else             grant = 4'b0000;
       2'b11 :
              if (req[3]) grant = 4'b1000;
         else if (req[0]) grant = 4'b0001;
         else if (req[1]) grant = 4'b0010;
         else if (req[2]) grant = 4'b0100;
         else             grant = 4'b0000;
     endcase
  end
```

Consider …

```
     if (req[0]) grant = 4'b0001;
else if (req[1]) grant = 4'b0010;
else if (req[2]) grant = 4'b0100;
else if (req[3]) grant = 4'b1000;
else             grant = 4'b0000;
```

This is equivalent to …

```
req[0] -> grant[0]
req[1] -> grant[1]
req[2] -> grant[2]
req[3] -> grant[3]
```

Or …

```
req[pointer_reg] -> grant[pointer_reg]
```

Each request and grant is independent of each other

Adding Nondeterminism

Free variable

```
module arbiter ...

  always @* begin
    case ( pointer_reg )
      2'b00 :
             if (req[0]) grant = 4'b0001;
        else if (req[1]) grant = 4'b0010;
        else if (req[2]) grant = 4'b0100;
        else if (req[3]) grant = 4'b1000;
        else             grant = 4'b0000;
      2'b01 :
             if (req[1]) grant = 4'b0010;
        else if (req[2]) grant = 4'b0100;
        else if (req[3]) grant = 4'b1000;
        else if (req[0]) grant = 4'b0001;
        else             grant = 4'b0000;
      2'b10 :
             if (req[2]) grant = 4'b0100;
        else if (req[3]) grant = 4'b1000;
        else if (req[0]) grant = 4'b0001;
        else if (req[1]) grant = 4'b0010;
        else             grant = 4'b0000;
      2'b11 :
             if (req[3]) grant = 4'b1000;
        else if (req[0]) grant = 4'b0001;
        else if (req[1]) grant = 4'b0010;
        else if (req[2]) grant = 4'b0100;
        else             grant = 4'b0000;
    endcase
  end
```

```
stopat pointer_reg
```

```
snip_driver pointer_reg
```

```
netlist cutpoint pointer_reg
```

Any starting point okay

Functionality modeled by the RTL

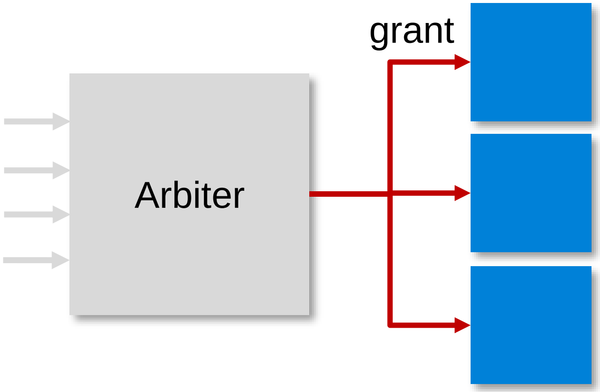
Reduced Cone of Influence

Free variable

```
module arbiter ...

  always @* begin
     case ( pointer_reg )
       2'b00 :
                if (req[0]) grant = 4'b0001;
           else if (req[1]) grant = 4'b0010;
           else if (req[2]) grant = 4'b0100;
           else if (req[3]) grant = 4'b1000;
           else             grant = 4'b0000;
       2'b01 :
                if (req[1]) grant = 4'b0010;
           else if (req[2]) grant = 4'b0100;
           else if (req[3]) grant = 4'b1000;
           else if (req[0]) grant = 4'b0001;
           else             grant = 4'b0000;
       2'b10 :
                if (req[2]) grant = 4'b0100;
           else if (req[3]) grant = 4'b1000;
           else if (req[0]) grant = 4'b0001;
           else if (req[1]) grant = 4'b0010;
           else             grant = 4'b0000;
       2'b11 :
                if (req[3]) grant = 4'b1000;
           else if (req[0]) grant = 4'b0001;
           else if (req[1]) grant = 4'b0010;
           else if (req[2]) grant = 4'b0100;
           else             grant = 4'b0000;
     endcase
  end
```

All upstream logic removed!




```
stopat grant
```

```
snip_driver grant
```

```
netlist cutpoint grant
```

Constrain behavior

```
assume property (
         $onehot0(grant));
```

How nondeterminism accelerates formal verification

Nondeterminism accelerates formal

- Reducing the cone of influence
- ➡ Simultaneous testing
- Faster property development
- Handling inconclusives
- Abstractions

DOULOS

Modeling with Free Variables

Undriven port

```
// Bind module
module ecc_assertions ( input logic [ 7:0] data,      // Free variable
                        input logic [12:0] ecc_from_dut, ... );
```

```
// ECC implementation
wire logic p1 = 1 ^ data[0] ^ data[1] ^ data[3] ^ data[4] ^ data[6];
wire logic p2 = 1 ^ data[0] ^ data[2] ^ data[3] ^ data[5] ^ data[6];
wire logic p4 = 1 ^ data[1] ^ data[2] ^ data[3] ^ data[7];
wire logic p8 = 1 ^ data[4] ^ data[5] ^ data[6] ^ data[7];

logic [12:0] expected_ecc;

assign expected_ecc[11:0] == { data[7], data[6], data[5], data[4],
                               p8, data[3], data[2], data[1], p4,
                               data[0], p2, p1 };
assign expected_ecc[12] = ^expected_ecc[11:0];      // Overall parity
```

```
assert property ( ecc_from_dut == expected_ecc );
```

Formal can test all values of data simultaneously!

Symbolic Constant

Symbolic constants do not change once initialized

```
bit [7:0] index;
```

Symbolic constant

```
assume property ( ##1 $stable(index) );
```

```
assert property ( request[index] |=> grant[index] );
```

All indices proven at the same time

Simplifies modeling and reduces analysis effort

How nondeterminism accelerates formal verification

Nondeterminism accelerates formal

- Reducing the cone of influence
- Simultaneous testing
- ➡ Faster property development
- Handling inconclusives
- Abstractions

Control Knobs

```
// Used by formal tool to select good or bad packets
enum bit  { FALSE = 0, TRUE = 1 } pkt_good;
```

Free variable

pkt_good acts as a knob in the constraints

```
property prop_sof(n);
    (packet[n].sof.sof == '0) and
    (packet[n].sof.unused == '0);
endproperty

// Start of frame constraint
property prop_pkt_sof(n);
   seq_kind(n,SOF) and
   seq_length(n,1) and
   if ( pkt_good )        prop_sof(n)
   else                not( prop_sof(n) );
endproperty
```

Knob negates constraint

Setting Knobs

```
// Sequence to indicate the design is parsing the packet
property pkt_parsing;
   !parse[*0:$] ##1 parse[*1:$] ##1 !parse;
endproperty
```

Control knob enables the formal target

```
ast_bad_pkt_marked_malformed :
  assert property ( not( pkt_good ) && pkt_sof && pkt_valid |->
                                malformed_signal within ( pkt_parsing ));
```

Simplifying Properties

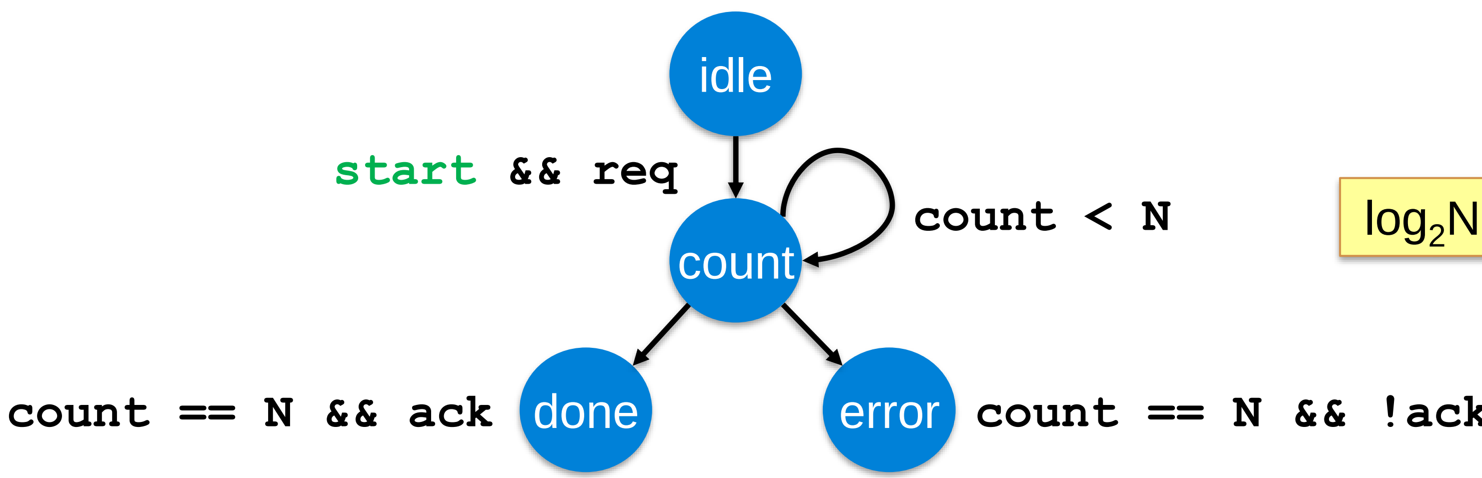
```
assert property ( req |-> ##N ack );
```

Synthesizes N number of flops – adds extra state space

2^N flops

State machines models property behaviors

- Focus only on interesting states
- Free variable `start` lets formal start at any time



$\log_2 N$ flops

Start Signals

Free variable

```
bit start;
enum bit [1:0] { IDLE, COUNT, DONE, ERROR } state;
bit [$clog2(N)-1:0] count;

always @( posedge clk or posedge rst )
    if ( rst ) begin
        state <= IDLE;
        count <= 0;
    end
    else
        case ( state )
            IDLE  : if ( start && req ) begin
                        state <= COUNT;
                        count <= 0;
                    end
            COUNT : if ( count == N ) begin
                        if ( ack ) state <= DONE;
                        else       state <= ERROR;
                    else           count <= count + 1'b1;
        endcase

assert property ( state != ERROR );
```

Free selection of any time

How nondeterminism accelerates formal verification

Nondeterminism accelerates formal

- Reducing the cone of influence
- Simultaneous testing
- Faster property development
- ➡ Handling inconclusives
- Abstractions

DOULOS

Deep State Space Search

```
always @(posedge reset or posedge clock)
  if (reset)
    count <= 0;
  else if (count < 16'hffff)
    count <= count + 1;
  else
    count <= 0;

assign en1 = (count == 16'h01ff);
assign en2 = (count == 16'h07ff);
assign en3 = (count == 16'h3fff);

always @(posedge clock)
begin: fail_eventually
  if (en1)
    array[address1] <= data_in; // Bounds check
  if (en2)
    array[address2] <= data_in; // Bounds check
  if (en3)
    array[address3] <= data_in; // Bounds check
end
```

Set proof depth = 5000

Fails at depth=512

Fails at depth=2048

Inconclusive at 5000

Cut Points

Set proof depth = 5000

```
always @(posedge reset or posedge clock)
  if (reset)
    count <= 0;
  else if (count < 16'hffff)
    count <= count + 1;
  else
    count <= 0;

assign en1 = (count == 16'h01ff);
assign en2 = (count == 16'h07ff);
assign en3 = (count == 16'h3fff);

always @(posedge clock)
begin: fail_eventually
  if (en1)
    array[address1] <= data_in; // Bounds check
  if (en2)
    array[address2] <= data_in; // Bounds check
  if (en3)
    array[address3] <= data_in; // Bounds check
end
```

Free variable

```
stopat count
```

```
snip_driver count
```

```
netlist cutpoint count
```

Fails at depth=1

Fails at depth=1

Fails at depth=1

How nondeterminism accelerates formal verification

Nondeterminism accelerates formal

- Reducing the cone of influence
- Simultaneous testing
- Faster property development
- Handling inconclusives
- ➡ Abstractions

Counters

Counters have a large state space

A counter abstraction can reduce the state space for formal

```
always @( posedge reset or posedge clock )
   if ( reset )
      count <= '0;
   else begin
      count <= count + 1;
```

Abstraction must provide functionality for:

Correct initialization

Counter behavior (including wrapping)

Counter Abstraction

Nondeterminism inserted using free variables

```
always @( posedge reset or posedge clock )
  if ( reset )
    count <= 0;
  else if ( count == MAX )
    count <= 0;
  else
    count <= count + 1;

logic [3:0] increment;

assume property (count == $past(count) + increment) % MAX);
```

Free variables

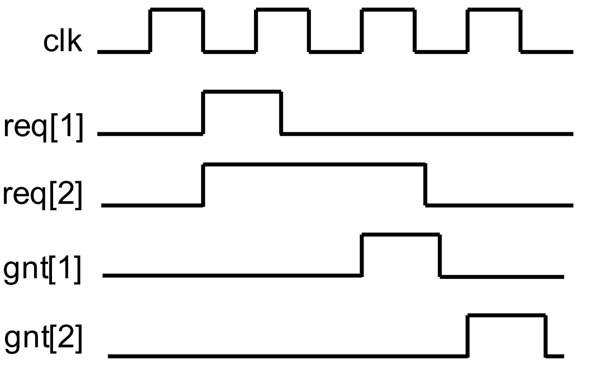
Add cut point

```
stopat count
```

```
snip_driver count
```

```
netlist cutpoint count
```

Priority Arbiter Example



Requirements:

j < k

req[j] has higher priority than req[k]

req[j] implies req[k] not granted until req[j] granted first

Simultaneous Checking

Free variables

```
bit [$clog2(N)-1:0] j, k;
```

```
// Symbolic constants
assume property ( ##1 $stable(j) && $stable(k) );

// Priority encoded
assume property ( j < k );

// Keep within range
assume property ( k < N );
```

All combinations of *j* and *k* checked simultaneously!

```
assert property ( req[j] |-> !gnt[k] s_until_with gnt[j] );
```

How nondeterminism accelerates formal verification

- Randomness in formal
- Nondeterminism accelerates formal
- ➡ Wrap-up

Summary

Nondeterminism accelerates formal by …

- Fewer properties => faster property development
- Formal modeling => simultaneous testing
- Reducing state space => faster state exploration
- Abstraction => deeper state exploration
- Randomness => exhaustive testing

Thank you for attending

We hope you found this information helpful!

DOULOS

Delivering KnowHow www.doulos.com

SoC Design & Verification

» SystemVerilog » UVM » Formal
» SystemC » TLM-2.0

FPGA & Hardware Design

» VHDL » Verilog » SystemVerilog
» Tcl » Xilinx » Intel FPGA (Altera)

Embedded Software

» Emb C/C++ » Emb Linux
» Yocto » RTOS » Security » Arm

Python & Deep Learning

DEEP LEARNING

SystemVerilog OUT OF THE BOX · UVM · SYSTEMC · yocto PROJECT PARTICIPANT · freeRTOS · SECURE EMBEDDED · arm APPROVED TRAINING PARTNER