

# Digital Verification for FPGA and ASIC Designers



Delivering KnowHow



# Digital Verification for FPGA and ASIC Designers





# Digital Verification for FPGA and ASIC Designers



Copyright © 2015-2023 by Doulos. All Rights Reserved

All intellectual property rights, including copyright, patents, design rights and know-how in or relating to the course or course materials provided or made available in connection with the course remain the sole property of Doulos Ltd or their respective owners and no copies may be made of course materials unless expressly agreed in writing by Doulos Ltd.

All trademarks acknowledged.

Doulos takes great care in developing and maintaining materials to ensure they are an effective and accurate medium for communicating design know-how. However, the information provided on a Doulos training course may be out of date or include omissions, inaccuracies or other errors. Except where expressly provided otherwise in agreement between you and Doulos, all information provided directly or indirectly through a Doulos training course is provided "as is" without warranty of any kind.

Doulos hereby disclaims all warranties with respect to this information, whether express or implied, including the implied warranties of merchantability, satisfactory quality and fitness for a particular purpose. In no event shall Doulos be liable for any direct, indirect, incidental special or consequential damages, or damages for loss of profits, revenue, data or use, incurred by you or any third party, whether in contract, tort or otherwise, arising from your access to, use of, or reliance upon information obtained from or through a Doulos training course. Doulos reserves the right to make changes, updates or corrections to the information contained in its training courses at any time without notice.

Doulos Limited  
Church Hatch, 22 Market Place,  
Ringwood, Hampshire, BH24 1AW, UK

Tel: +44 (0) 1425 471223

Email: [info@doulos.com](mailto:info@doulos.com)

Doulos  
6203 San Ignacio Avenue, Suite 110,  
San Jose, CA 95119, USA

Tel: 1-888-GO DOULOS

Email: [info.usa@doulos.com](mailto:info.usa@doulos.com)

[www.doulos.com](http://www.doulos.com)



# Contents

Contents.....	7
Current Verification Landscape.....	9
Verification Approaches .....	9
Simulation and Testbenches.....	13
Coverage .....	20
Formal Verification .....	23
Class-Based SystemVerilog Verification.....	26
What is SystemVerilog? .....	26
SystemVerilog Classes .....	31
Virtual Interfaces .....	38
Constraints and Functional Coverage .....	41
Universal Verification Methodology (UVM).....	44
What is UVM? .....	44
UVM Hello World.....	48
DUT Interface.....	53
Sequencer-Driver Communication .....	57
Formal Verification for Non-Specialists .....	62
Learning to use Formal .....	62
Writing Properties.....	63
Tackling State Space .....	67
Under-constraining versus Over-constraining .....	71
Using Formal.....	76
Conclusions and Recommendations.....	77



## Notes



## Current Verification Landscape

### Verification Approaches

### Current Verification Landscape

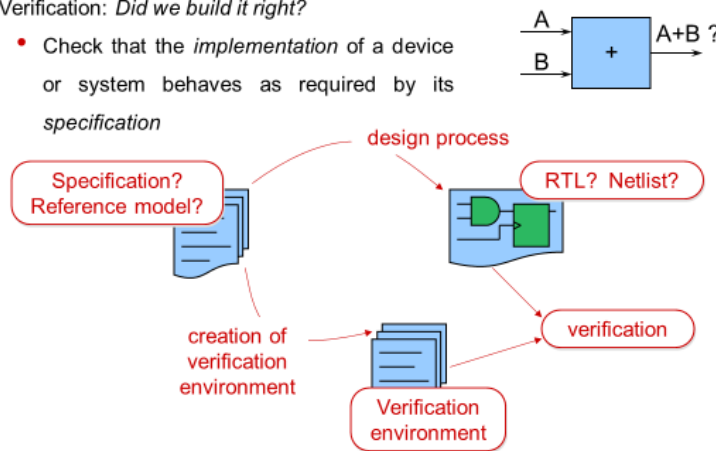
➔ Verification Approaches

- Simulation and Testbenches
- Coverage
- Formal Verification

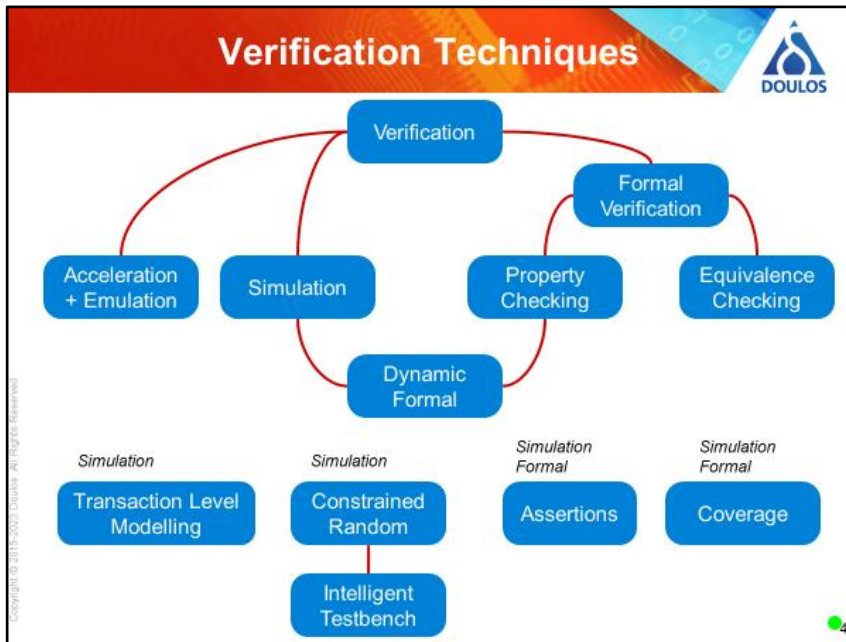
2

### What is Verification?

- Verification: *Did we build it right?*
  - Check that the *implementation* of a device or system behaves as required by its *specification*



3

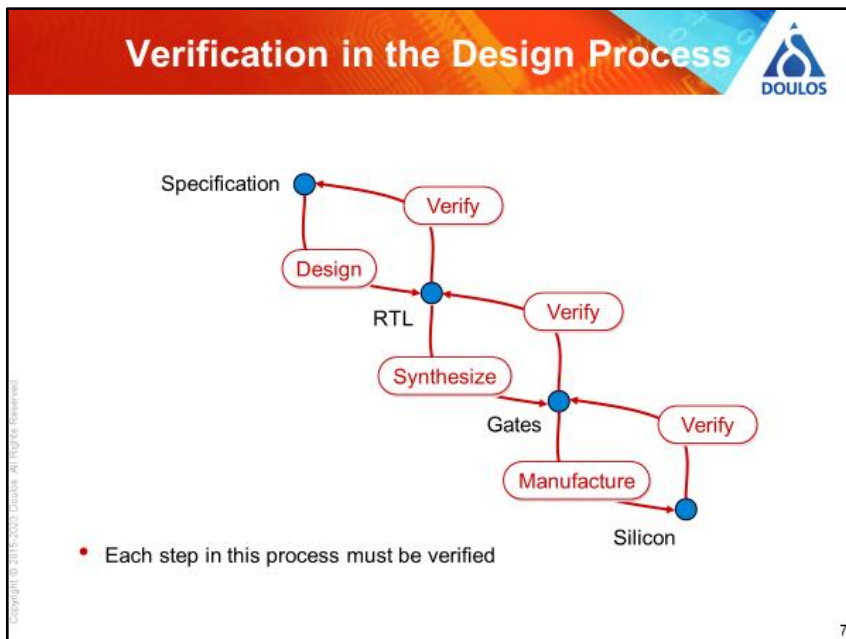
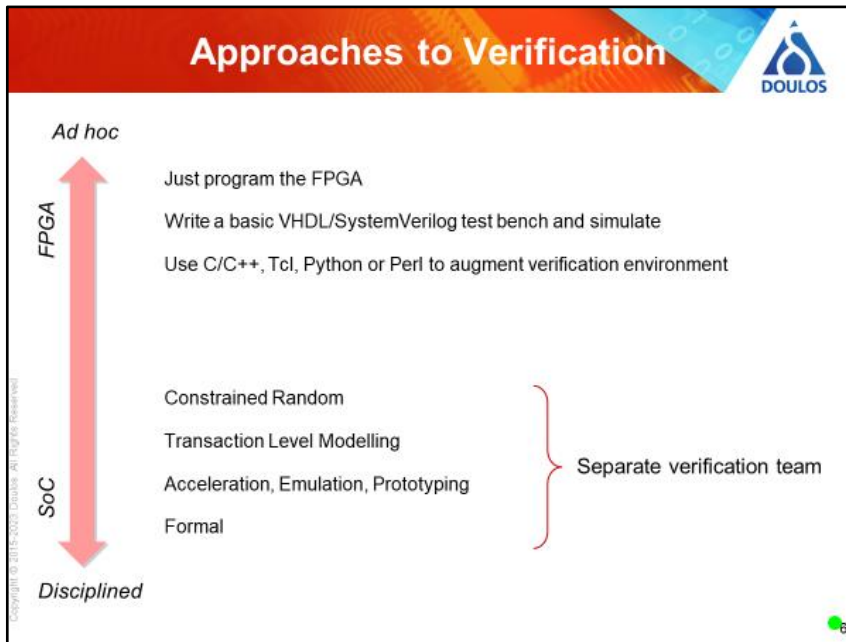


## Languages for Verification

*List has not changed for years...*

	<b>Crude Caricature</b>
<ul style="list-style-type: none"> <li>• IEEE 1076 VHDL</li> <li>• IEEE 1850™ PSL</li> </ul>	<i>FPGA, RTL, Europe, Mil-Aero</i>
<ul style="list-style-type: none"> <li>• IEEE 1364 Verilog</li> <li>• <b>IEEE 1800™ SystemVerilog</b></li> </ul>	<i>ASIC, RTL, USA/RoW</i>
	<i>Hardware verification</i>
	<span style="border: 1px solid red; border-radius: 10px; padding: 2px;">Most Popular choice</span>
<ul style="list-style-type: none"> <li>• IEEE 1647™ e</li> </ul>	<i>Advanced Hardware verification</i>
<ul style="list-style-type: none"> <li>• ISO/IEC 14882 C++</li> <li>• IEEE 1666™ SystemC</li> </ul>	<i>Modelling, verification</i>
	<i>Virtual hardware prototypes for S/W dev</i>
<ul style="list-style-type: none"> <li>• Tcl/Tk, Python, Perl</li> </ul>	<i>Scripting</i>

Copyright © 2015-2023 Doulos. All Rights Reserved



## Verification Plan



- What are we going to verify? Requirements, objectives
- How are we going to do it? Design of test stimulus
- How do we measure success? Observations


The *Verification Plan* specifies the verification tasks and effort

- If it's specified it should be verified!
- Identify an appropriate way to test each feature or statement in the specification
  - *directed or constrained-randomised* testing
- Create constrained random stimulus that will execute that feature and the data 'self checking' (aka scoreboard) is correctly implemented by the design

Copyright © 2015-2023 Doulos. All Rights Reserved

8

## Example Verification Plan



Specification

- The NBG output pin will reflect the status of the internal FAIL register bit
- A checksum calculated using the CCITT-16 polynomial is appended

Test descriptions

1. Write '1' to FAIL register
2. Check that NBG goes to '1' within 2 clocks
3. -----

1. Cover NBG signal value wrt FAIL register bit.
2. Cover point for checksum value good and bad using CCITT-16

1. Change the FAIL bit value in the register .
2. Generate and send data using good and bad checksum values

- In this example, one specification part is verified using a defined test
  - This could be done by simulation or by formal methods
- For the second part, (constrained) random data is generated

Copyright © 2015-2023 Doulos. All Rights Reserved

9

## Linting Tools

- Locating errors or potential errors in HDL code can save a lot of verification effort later
- Simulators and formal tools should find errors ... eventually
- A *Linting Tool* finds common errors quickly and automatically
- Example:

```
always @(Select)
  if (Select)
    Y= A;
  else
    Y=B;
```

- Verilint
- HAL
- LEDA
- ...

Warning: Incomplete event list

Copyright © 2015-2023 Doulos. All Rights Reserved

10


## Simulation and Testbenches

### Current Verification Landscape


- Verification Approaches
  - ➔ Simulation and Testbenches
- Coverage
- Formal Verification

11


## Simulation



- Execute a model of the system
- A simulation is only as good as the model
  - The more accurate the model, the longer the simulation time
- Cannot simulate everything – not enough time



Accuracy




Speed

- Widely used:
  - simulation is intuitively attractive, looks like the real thing
  - mature, familiar tools
  - excellent debugging

Copyright © 2015-2023 Doulos. All Rights Reserved

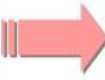
12

## Simulating a digital system



Test vectors

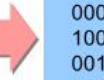
```
11001001
01001010
00001001
01110110
01100110
01001001
01001110
```



VHDL

Design Under Test

Verilog



Output vectors

```
000010
010011
000010
100100
001000
110010
000011
```

- #1 tool for functional verification
- Vectors must be created:
  - write a program to do it ("directed")

OR

  - compute them using constrained-random techniques (requires automated checking of output)

Copyright © 2015-2023 Doulos. All Rights Reserved

13

## Results from Simulation



- Only an approximation of reality
- Semantics of HDL and simulator define simulation behaviour
- Simulation requires stimulus
  - quality of test is determined by quality of stimulus

Results are determined by both the design and the stimulus

- Response must be verified against expected behaviour
- General problem: Simulators are never fast enough!

Copyright © 2015-2023 Doulos. All Rights Reserved

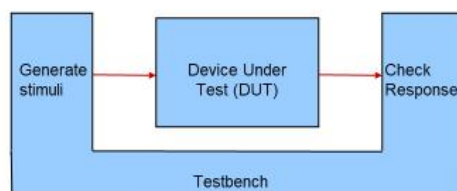
14

## Basic Testbench

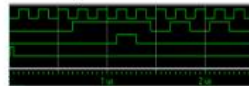


Often written by the designer  
Unstructured grey box test

```
#10 selA = 1'b1;
#10 A = 1'b0;
#10 selA = 1'b0;
```




Inspect waveform?



Copyright © 2015-2023 Doulos. All Rights Reserved

15

## Testbenches




- A *testbench* sends stimuli to the Device under Test (DUT) and collects responses from the DUT
  - Usually designed to be *self-checking*, i.e. it automatically checks the DUT's responses for correctness
- A testbench is therefore the environment seen by the DUT
  - The testbench must provide all signals needed for the DUT to operate
- The testbench can be written in:
  - the same HDL as the DUT
  - another HDL, or
  - a specialised Hardware Verification Language (HVL)
- Testbenches can use the entire range of the language – no need to stick to a synthesis subset

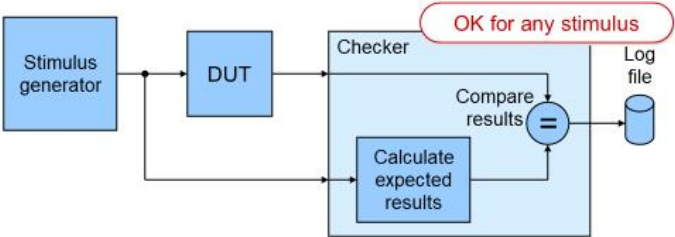
Copyright © 2015-2023 Doulos. All Rights Reserved

16

## Structured Testbench



- Stimulus can be read from a file or can be created using constrained-random generation
- Whatever stimulus is applied, the DUT's output should be checked automatically
- Prefer monitoring/checking processes that work correctly for *any* stimulus



```

            graph LR
                SG[Stimulus generator] --> DUT[DUT]
                DUT --> Checker
                subgraph Checker
                    CER[Calculate expected results]
                    CR[Compare results]
                    CR --> OK(OK for any stimulus)
                end
                SG --> CER
                OK --> LF[(Log file)]
            
```

Copyright © 2015-2023 Doulos. All Rights Reserved

17



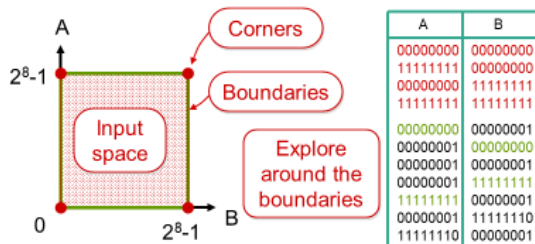
## Boundary Conditions & Corner Case



- Exhaustive testing is impractical, so which tests to include?
- Example - an arithmetic function:

```
entity Arithmetic is
  port (
    A, B : in  UNSIGNED(7 downto 0);
    F     : out UNSIGNED(7 downto 0));
end entity Arithmetic;
```

- Verification engineer's experience is important in choosing corner cases
- When using formal verification, the story is very different - all cases explored automatically



A	B
00000000	00000000
11111111	00000000
00000000	11111111
11111111	11111111
00000000	00000001
00000001	00000000
00000001	00000001
00000001	11111111
11111111	00000001
00000001	11111110
11111110	00000001

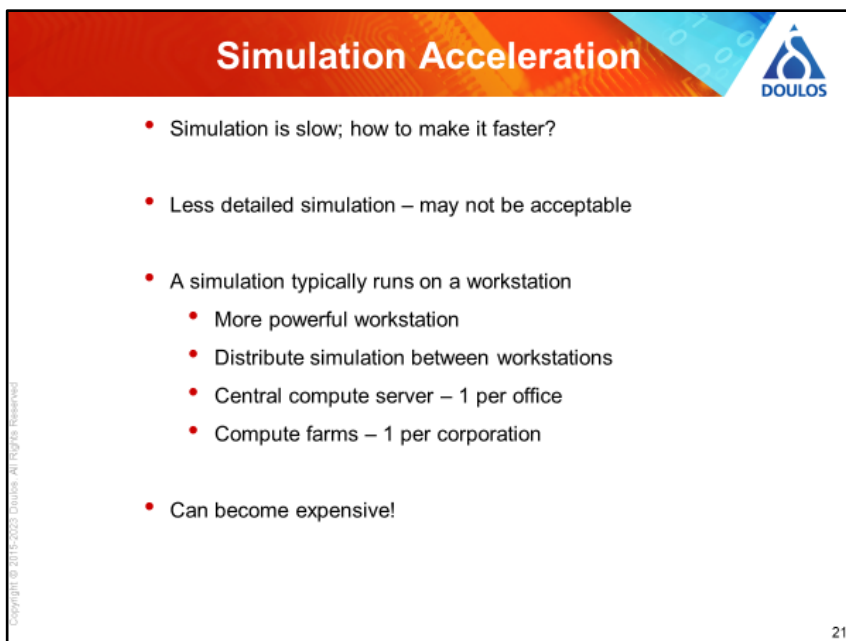
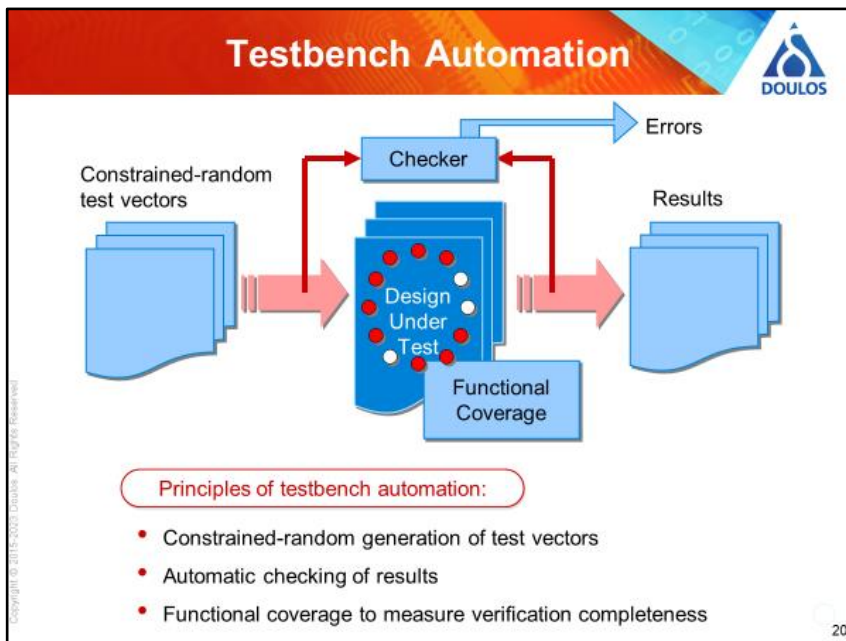
18

## Constrained Random Tests




- Writing many different tests is difficult "Directed" tests
  - In the worst case, need every combination of inputs
  - Choose a representative sample – how?
  - Unconscious bias towards good data – may ignore unlikely but fatal combinations
- Random test generation is easy
  - Random number generation built in to most languages
  - But... many random combinations should not occur
- Constrained random generation
  - Random, but subject to certain constraints
- Example
  - Meaningful sequence of opcodes, but random data and random addresses within a certain range

19



## Modelling Levels




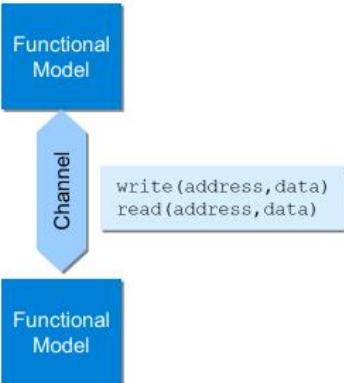
- If the complete DUV is modelled at the lowest level, the simulation will be slow
- Ideally, model each part of the DUV only in enough detail to verify that part of the specification that's being simulated
- Example
  - If we model a data transfer operation, every bit change on every pin will generate a simulation event
- Solution
  - Model the data transfer as a single transaction – one event
- This is *Transaction Level Modelling (TLM)*

Copyright © 2015-2023 Doulos. All Rights Reserved

22

## Transaction Level Modelling





```

graph TD
    FM1[Functional Model] <-->|Channel| FM2[Functional Model]
    subgraph Channel
        direction TB
        C1[write(address, data)]
        C2[read(address, data)]
    end
  
```

- 100+ X faster simulation!


Copyright © 2015-2023 Doulos. All Rights Reserved

23

## Coverage

### Current Verification Landscape


- Verification Approaches
- Simulation and Testbenches
- ➔ Coverage
- Formal Verification



24

### Coverage

- "Coverage" is used in different ways –
- Usual English meaning
  - "Have we covered everything in the specification?"
- Technical terms
  - *Code Coverage* – how much of the code have we exercised during simulation?
  - *Functional coverage* – how much (expressed as a percentage) of the functionality described by the specification have we exercised during simulation?



25

## Code Coverage

**Test vectors**

```
11001001
01001010
00001001
01110110
01100110
01001001
01001110
```

**Device under test**

```

20 if Reset = '1' then
1   Cnt <= "00000000";
19 elsif Rising_edge(Clock) then
9   if Enable = '0' then
1     null;
8   elsif Load = '1' then
2     Cnt <= Unsigned(Data);
6   else
6     if UpDn = '1' then
6       Cnt <= Cnt + 1;
0     else
0       Cnt <= Cnt - 1;
0     end if;
0     end if;
0   end if;
0   end if;
0   end if;

```

**Output vectors**

```
000010
010011
000010
100100
001000
110010
000011
```

**Execution count** (points to line 20)

**Not yet executed – might be a bug here!** (points to line 20)

**This is line coverage** (points to the code block)

- Doesn't prove correctness!

26

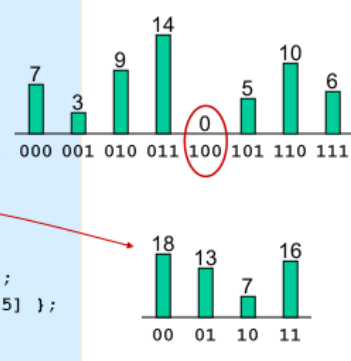
## Sample-Based Functional Coverage

```

class instruction;
bit [2:0] opcode;
bit [1:0] mode;
shortint unsigned data;

covergroup cg @(posedge clk);
coverpoint opcode;
coverpoint mode;
coverpoint data {
  bins tiny [8] = { [0:7] };
  bins moderate[8] = { [8:255] };
  bins huge [8] = { [256:65535] };
}
endgroup
...
endclass: instruction

```

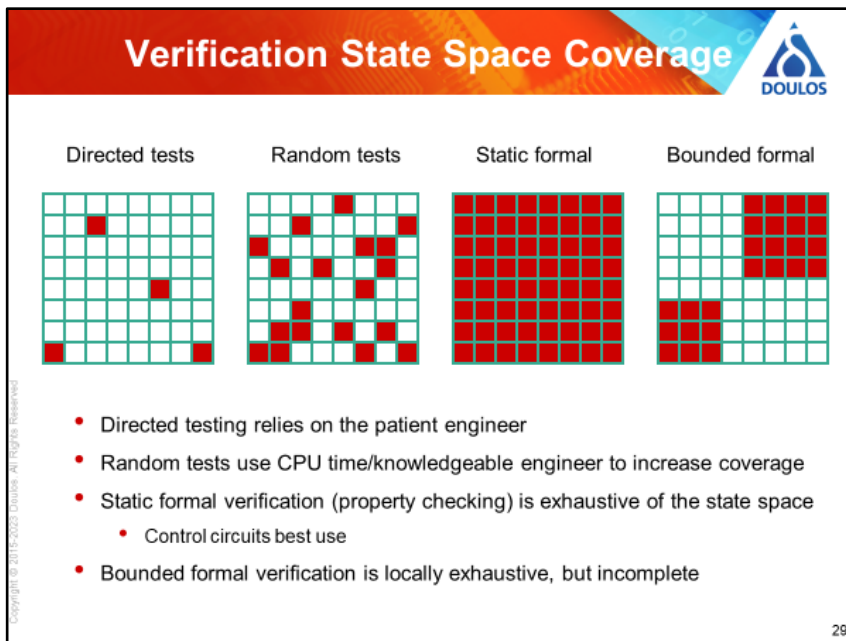
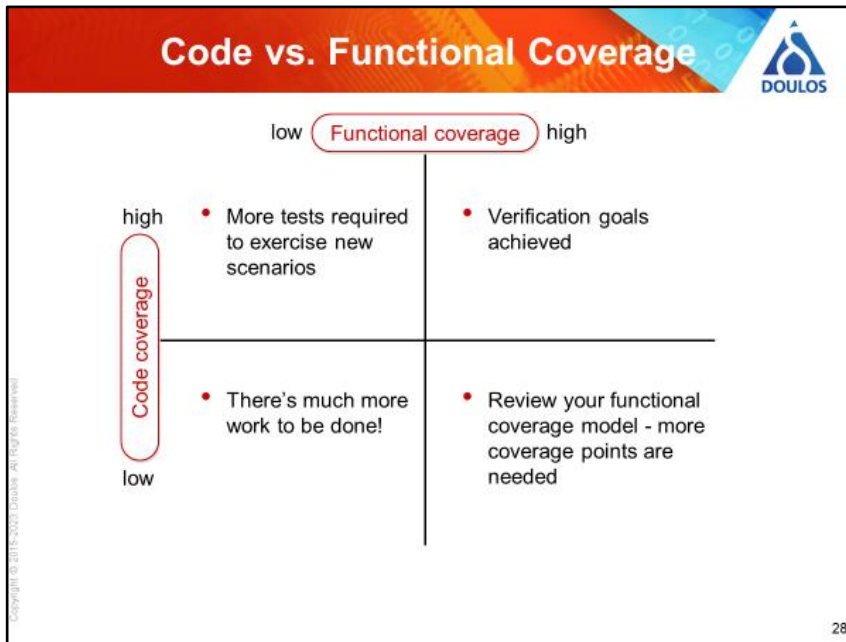


Opcode	Count
000	7
001	3
010	9
011	14
100	0
101	5
110	10
111	6

Mode	Count
00	18
01	13
10	7
11	16

**SystemVerilog**


27



## Formal Verification

### Current Verification Landscape


- Verification Approaches
- Simulation and Testbenches
- Coverage
- ➔ Formal Verification



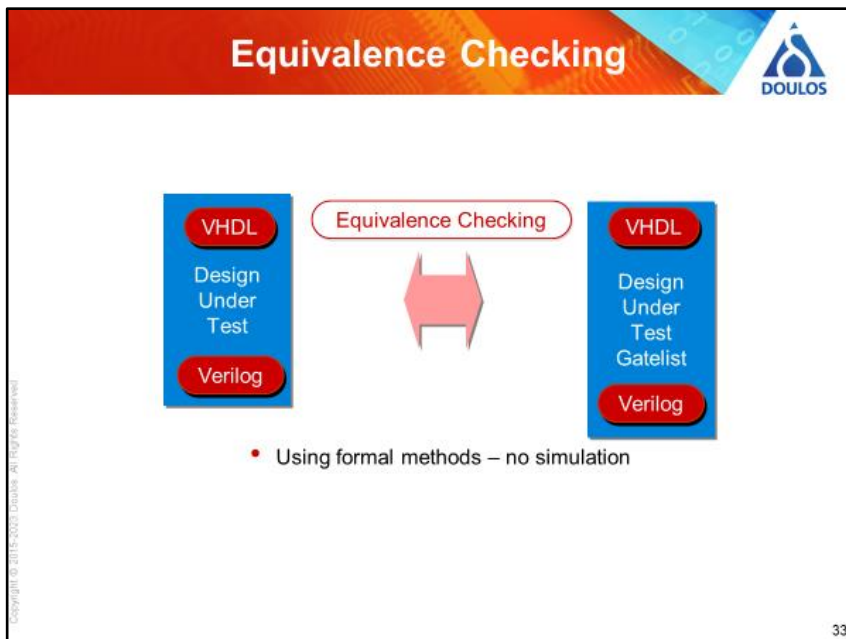
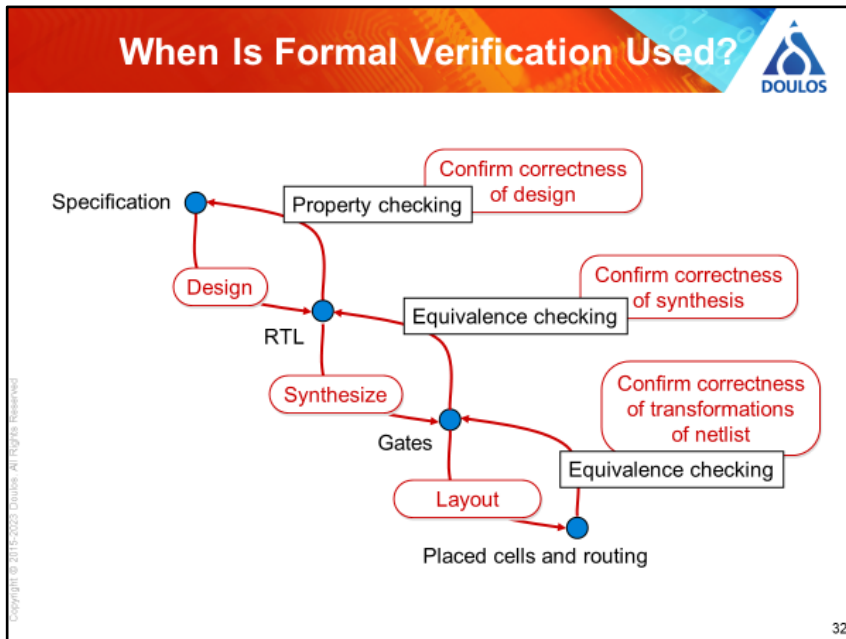
30

### Formal Verification

- The alternative to simulation is mathematical proof
  - Usually known as *Formal Methods*
  - Covers a wide variety of techniques
- Formal methods do not need stimuli – can be considered to be exhaustive
  - But not everything can be checked – best suited to state machines
- Why not always use formal methods?
  - State explosion problem – every possible sequence of states
- Two basic techniques:
  - *Equivalence checking* – do two versions of the system have the same functionality
  - *Property checking* – does a system satisfy certain properties?



31






## Property Checking

VHDL  
Design  
Under  
Test  
Verilog

Property Checking



Properties  

assert A > B

Property languages

- Using formal methods – no simulation
- Exhaustive state space coverage

Copyright © 2015-2023 Doulos. All Rights Reserved

34

## Assertions

- An *assertion* is an instruction to check a *property* of the design
- Can be checked by a simulator or by a property checker

Simple Checks

```
assert output1 > output2
```

- Equivalent to combinational logic

Complex Temporal Checks

```
when input1 rises check that output1 > output2 after 2 clocks
```

- Equivalent to a state machine

Copyright © 2015-2023 Doulos. All Rights Reserved

35


## Class-Based SystemVerilog Verification

### What is SystemVerilog?

### Class-Based SystemVerilog Verification

➔ What is SystemVerilog?

- SystemVerilog Classes
- Virtual Interfaces
- Constraints and Functional Coverage



36

### What is SystemVerilog

- The world's first HDVL, Hardware Design and Verification Language
- IEEE Std 1364-2005 Verilog and
- IEEE Std 1800-2005 SystemVerilog

merged to form

- IEEE Std 1800-2009 SystemVerilog
- IEEE Std 1800-2012 SystemVerilog
- IEEE Std 1800-2017 SystemVerilog

- **SystemVerilog RTL**, aka concise RTL
- **SystemVerilog Assertions**, aka SVA
- **SystemVerilog Testbench**, or class-based verification



37

## SystemVerilog Language Features

**RTL + programming**

- C-style data types & control - enum, struct, typedef, ++, break, return
- Synthesis-friendly "concise" RTL notation
- Packages
- Interfaces

**Assertions**

- SystemVerilog Assertions

**Testbench**

- Clocking blocks (synchronization between DUT and test bench)
- Object-oriented programming - classes
- Constrained random stimulus generation
- Functional coverage
- Dynamic processes, dynamic arrays, queues, mailboxes, semaphores

- Direct Programming Interface (DPI) - calling C from SystemVerilog
- Extensions to VPI

Copyright © 2015-2023 Doulos. All Rights Reserved

38

## Caveats

- C-like control constructs and data types
- Concise RTL
- VHDL-like package and import
- Assertions

} A better Verilog

- Non-portable constructs

} Ill-defined

- Classes
- Constraints and coverage based on classes
- Built-in types - strings, queues, maps
- Virtual interfaces


} Class-based verification

Used by standard verification methodologies

Copyright © 2015-2023 Doulos. All Rights Reserved

39

## 4-State and 2-State Types



- 4-state types
 


<i>Signed</i>	<i>Unsigned</i>	<i>Width</i>
logic signed	logic	1 bit
logic signed [n:m]	logic [n:m]	N bits
- 2-state types (variables only, not wires)
 

<i>Signed</i>	<i>Unsigned</i>	<i>Width</i>
bit signed	bit	1 bit
bit signed [n:m]	bit [n:m]	N bits
byte	byte unsigned	8 bits
shortint	shortint unsigned	16 bits
int	int unsigned	32 bits
longint	longint unsigned	64 bits

Copyright © 2015-2023 Doulos. All Rights Reserved

40

## Struct



- Aggregate of dissimilar data items, just like C
- Best used with typedef

```

typedef struct {
    bit b;
    int i;
    logic [7:0] v;
} mystruct_t;
    
```

```

mystruct_t s;
s.b = 1;
s.i = -8;
s.v = 8'hff;

s = '{1, -8, 8'hff};
    
```

Variable

Assignment pattern

Copyright © 2015-2023 Doulos. All Rights Reserved

41

## Interfaces

- Simple interface = bundle of wires/vars

```

interface APB;
  logic PCLK, PSEL, PENABLE, PWRITE;
  logic [15:0] PADDR;
  logic [31:0] PWDATA;
  logic [31:0] PRDATA;
endinterface
  
```

```

module Master (APB iport, ...);
  ...
endmodule
  
```

Interface port
Must use ANSI-style port list

Copyright © 2015-2023 by Doulos. All Rights Reserved

42

## Immediate and Concurrent Assertion

- Procedural assertion – sampled procedurally

```

always ...
  assert ( EXPRESSION );
  
```

Ordinary SystemVerilog expression

- Concurrent assertion – condition is usually sampled on clock edge

```

assert property ( @(posedge Clock) CONDITION );
  
```

SystemVerilog property

- Condition is only tested when pre-condition has been matched

```


assert property (
  @(posedge Clock) PRECONDITION |-> CONDITION );
  
```

SystemVerilog sequence
Implication operator

Copyright © 2015-2023 by Doulos. All Rights Reserved

43

## Concurrent Assertions



- Check or prove the property
 

label: **assert property** (PROPERTY) ACTION\_BLOCK;

Important
- Collect functional coverage information
 


label: **cover property** (PROPERTY) STATEMENT;
- Make the property an assumption for formal
 

label: **assume property** (PROPERTY);

Copyright © 2015-2023 Doulos. All Rights Reserved

44

## Temporal Behaviour



- Properties and sequences describe temporal behaviour
- "Temporal" means the sequence spans more than one clock cycle

Concurrent assertion

Sequences

```

assert property (
  @(posedge Clock) (a ##1 b) |-> (d ##1 e)
);
                
```

Property

- Termination mid-way through matching a sequence
  - (Discharges property's obligation to hold for PROPERTY)

Expression

```

assert property (
  @(posedge Clock) disable iff (TERMINATE) PROPERTY );
                
```

Termination operator

Copyright © 2015-2023 Doulos. All Rights Reserved

45

## SystemVerilog Classes

### Class-Based SystemVerilog Verification

- What is SystemVerilog?
- ➔ SystemVerilog Classes
- Virtual Interfaces
- Constraints and Functional Coverage



46

## SystemVerilog Classes



```

package Bus_pkg;
typedef logic [15:0] T_addr;
typedef logic [15:0] T_data;
typedef enum bit (dir_Rd, dir_Wr) T_dir;
class Bus_trans;
    int ID;
    T_dir dir;
    T_addr addr;
    T_data data;
    function void print;
        string kind = (dir==dir_Rd) ? "Read" : "Write";
        $display("%s cycle %0d: A=%h, D=%h",
                kind, ID, addr, data);
    endfunction : print
endclass : Bus_trans

endpackage : Bus_pkg

```

Always put classes in a package

Class defines a transaction object


Data members or class properties

Method

Copyright © 2015-2023 by Doulos. All Rights Reserved

47

## Object = Instance of Class



```

module use_Bus_trans;
  import Bus_pkg::*;
  Bus_trans t1, t2;
  initial begin
    ...
  end

```

**references**


t1

t2

- Variables of class type store *references* (handles) to real objects
  - Initialised to `null` (reference to no object)

Copyright © 2015-2023 Doulos. All Rights Reserved

## Object = Instance of Class



```

module use_Bus_trans;
  import Bus_pkg::*;
  Bus_trans t1, t2;
  initial begin
    t1 = new;
    ...
  end

```

**references**

t1

t2

**object**

ID	0
dir	dir_Rd
addr	xxxx
data	xxxx

- Variables of class type store *references* (handles) to real objects
  - Initialised to `null` (reference to no object)
- Create objects using `new` - data members get their usual default values

Copyright © 2015-2023 Doulos. All Rights Reserved

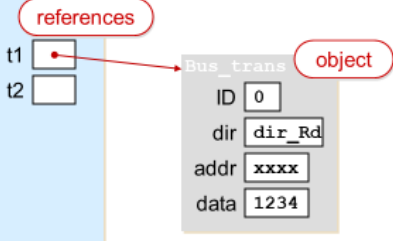


## Object = Instance of Class

```

module use_Bus_trans;
import Bus_pkg::*;
Bus_trans t1, t2;
initial begin
    t1 = new;
    t1.data = 16'h1234;
...

```



The diagram illustrates the relationship between a reference variable and an object. On the left, a variable `t1` is shown with a red dot inside a box, labeled "references". An arrow points from this dot to a larger box representing a `Bus_trans` object, labeled "object". The object contains four fields: `ID` with value `0`, `dir` with value `dir_Rd`, `addr` with value `xxxx`, and `data` with value `1234`.

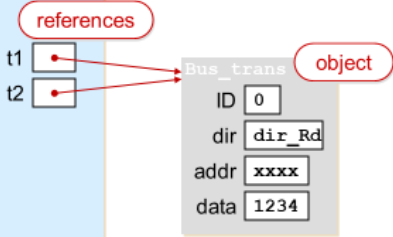
- Variables of class type store *references* (handles) to real objects
  - Initialised to `null` (reference to no object)
- Create objects using `new` - data members get their usual default values
- Access data members, and call methods, in existing objects using dot notation

## Object = Instance of Class

```

module use_Bus_trans;
import Bus_pkg::*;
Bus_trans t1, t2;
initial begin
    t1 = new;
    t1.data = 16'h1234;
    t2 = t1;
...


```



The diagram illustrates the relationship between two reference variables and a single object. On the left, two variables `t1` and `t2` are shown, each with a red dot inside a box, labeled "references". Arrows from both dots point to a single larger box representing a `Bus_trans` object, labeled "object". The object contains four fields: `ID` with value `0`, `dir` with value `dir_Rd`, `addr` with value `xxxx`, and `data` with value `1234`.

- Variables of class type store *references* (handles) to real objects
  - Initialised to `null` (reference to no object)
- Create objects using `new` - data members get their usual default values
- Access data members, and call methods, in existing objects using dot notation

## Object = Instance of Class



```

module use_Bus_trans;
  import Bus_pkg::*;
  Bus_trans t1, t2;
  initial begin
    t1 = new;
    t1.data = 16'h1234;
    t2 = t1;
    t2.data = 16'habcd;
  ...
  
```

**references**

t1 •

t2 •


**object**

Bus\_trans  
 ID 0  
 dir dir\_Rd  
 addr xxxx  
 data abcd

- Variables of class type store *references* (handles) to real objects
  - Initialised to `null` (reference to no object)
- Create objects using `new` - data members get their usual default values
- Access data members, and call methods, in existing objects using dot notation

Copyright © 2015-2023 Doulos. All Rights Reserved

## Object = Instance of Class



```

module use_Bus_trans;
  import Bus_pkg::*;
  Bus_trans t1, t2;
  initial begin
    t1 = new;
    t1.data = 16'h1234;
    t2 = t1;
    t2.data = 16'habcd;
    t1.print();
  ...
  
```

**references**

t1 •

t2 •

**object**

Bus\_trans  
 ID 0  
 dir dir\_Rd  
 addr xxxx  
 data abcd

**Method call** Read cycle #0: A=xxxx, D=abcd

- Variables of class type store *references* (handles) to real objects
  - Initialised to `null` (reference to no object)
- Create objects using `new` - data members get their usual default values
- Access data members, and call methods, in existing objects using dot notation
  - Methods act on the object through which they were called

Copyright © 2015-2023 Doulos. All Rights Reserved

## Initializing Objects



```
module use_Bus_trans;
  import Bus_pkg::*;
  Bus_trans t3 = new;
```

Create object with default initial values

- **new** allocates memory and calls default constructor

```
class Bus_trans;
  ...
  function new;
    addr = 0;
    dir = dir_Wr;
    $write("Created new ");
    print();
  endfunction : new
  ...
```

Explicit constructor new. No return type

Call a method from within the class

print myself

54

## Constructor Arguments



- Like any function in SystemVerilog, constructors may have arguments

```
function new (T_dir direction);
  addr = 0;
  dir = direction;
  $write("Created new "); print();
endfunction : new
```

```
Bus_trans t4 = new;
```

ERROR

```
Bus_trans t4 = new(dir_Rd);
```

Read

- Arguments may have default values (no overloading, though)

```
function new (T_dir direction = dir_Rd); ...
```

```
Bus_trans t5 = new;
```


Read

```
Bus_trans t6 = new(dir_Wr);
```

Write

55

## Randomized Data Members



```

class Bus_trans;
  static int next_ID;
  const int ID;
  rand T_dir dir;
  rand T_addr addr;
  rand T_data data;
  function new;
    ID = next_ID++;
  endfunction : new
  function void print; ...
endclass : Bus_trans

Bus_trans tR;
repeat (3) begin
  tR = new;
  void'( tR.randomize() );
  tR.print();
end
    
```

**any data member can be declared rand**

**unique serial number**

**randomize an existing object**

Bus_trans	
ID	0
dir	dir_Wr
addr	35e7
data	4a8f


  

Bus_trans	
ID	1
dir	dir_Wr
addr	b267
data	04e3

Write cycle #0: A=35e7, D=4a8f  
Write cycle #1: A=b267, D=04e3

Copyright © 2015-2023 Doulos. All Rights Reserved

## Randomized Data Members



```

class Bus_trans;
  static int next_ID;
  const int ID;
  rand T_dir dir;
  rand T_addr addr;
  rand T_data data;
  function new;
    ID = next_ID++;
  endfunction : new
  function void print; ...
endclass : Bus_trans

Bus_trans tR;
repeat (3) begin
  tR = new;
  void'( tR.randomize() );
  tR.print();
end
    
```

**any data member can be declared rand**

**randomize an existing object**

Bus_trans	
ID	0
dir	dir_Wr
addr	35e7
data	4a8f

Bus_trans	
ID	1
dir	dir_Wr
addr	b267
data	04e3

Bus_trans	
ID	2
dir	dir_Rd
addr	1040
data	93c2

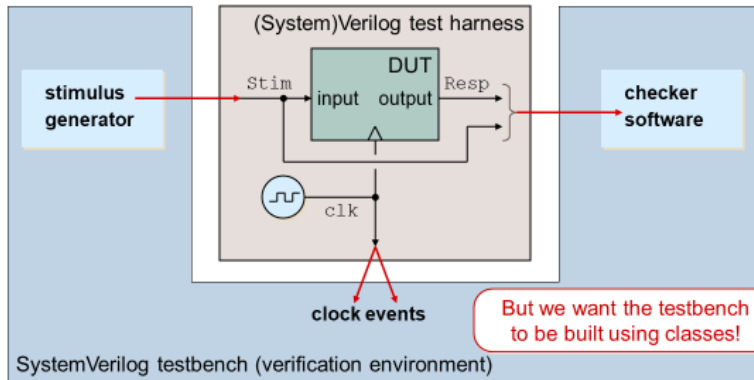
Write cycle #0: A=35e7, D=4a8f  
Write cycle #1: A=b267, D=04e3  
Read cycle #2: A=1040, D=93c2

Copyright © 2015-2023 Doulos. All Rights Reserved

## Test Harness and Testbench



- Test harness is a module containing:
  - the DUT instance and connections to its ports
  - clock generator and other support structures



58

## Lifetime and Persistence



- Module, interface and program instances:
  - created at elaboration, before simulation begins
  - hierarchy structure controlled by parameters
  - structure/instances cannot be changed dynamically
- Objects of class type:
  - created dynamically, during simulation, using new
  - structure controlled by run-time activity
  - can be created and destroyed at any time

verification environment

transaction data objects

- typically constructed at time zero
- structure probably remains unchanged throughout simulation
- created in large numbers during the simulation
- destroyed after use (unless logged)

59

## Creating the Testbench

```

module TB_top;
import TB_pkg::*;
TB_env tb;
initial begin
    tb = new;
    tb.run();
end
endmodule : TB_top
        
```

```

module harness;
logic Stim, Resp;
bit clk;
Sys_Top DUT (.*);
...
endmodule
        
```

**testbench object**

```

class TB_env;
...
task drive_Stim(input bit data);
    @(posedge harness.clk)
    harness.Stim <= data;
endtask
...
        
```

**Test harness module**

- Our entire testbench class is hard-coded for the name of the test harness!

Copyright © 2015-2023 Doulos. All Rights Reserved

60

## Virtual Interfaces

### Class-Based SystemVerilog Verification

- What is SystemVerilog?
- SystemVerilog Classes
- ➔ Virtual Interfaces
- Constraints and Functional Coverage

61

## Virtual Interface

```


class TB_env;
  virtual TB_hook hook;
  function new(virtual TB_hook h);
    hook = h;
    ...
  endfunction : new
  task drive_Stim(input bit data);
    @(posedge hook.clk)
      hook.Stim <= data;
  endtask
  ...

```

```

interface TB_hook;
  logic Stim, Resp;
  bit clk;
endinterface

```



**testbench object**

```

interface TB_hook
  clk _____
  Stim _____
  Resp _____

```

interface instance

- Where is this instantiated?
- How does it link to the DUT?

- Testbench class now OK for *any* instance of a TB\_hook interface
- Link TB object to interface instance at runtime

62

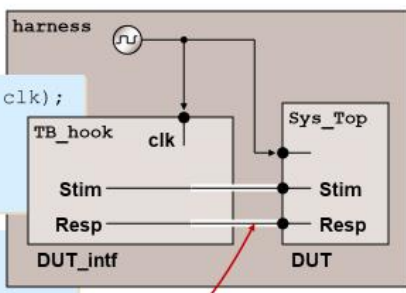
## Building a test harness

```

interface TB_hook (input bit clk);
  wire Stim, Resp;
  ...
endinterface

module harness;
  bit clk;
  TB_hook DUT_intf (.clk);
  Sys_Top DUT (
    .clk (clk),
    .Stim (DUT_intf.Stim),
    .Resp (DUT_intf.Resp),
    ...
  )
endmodule

```



Hierarchical connection


Harness contents

- DUT instance
- DUT connections
- clock generator

Interface contains all signals required by testbench

63

## Connecting the virtual interface



```

class TB_env;
    virtual TB_hook V;
    function new (virtual TB_hook V, ...);
        this.V = V;
        ...
    endfunction
    ...
endclass

module TB_top;
    TB_env tb;
    ...
    initial begin
        tb = new(harness.DUT_intf, ...);
    end
endmodule

module harness;
    bit clk;
    TB_hook DUT_intf (.clk);
    Sys_Top DUT (
        ...
    )
endmodule
    
```

constructor

Choice of interface type


test harness

Choice of instance

Copyright © 2015-2023 Doulos. All Rights Reserved

64

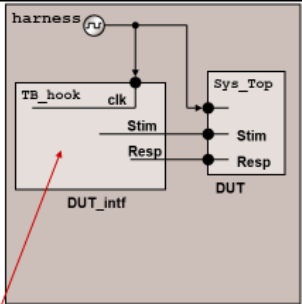
## Testbench Static Structure



- Test case (TB\_top) and test harness are the only static instances

```

module TB_top;
    import TB_pkg::*;
    TB_env tb;
    initial begin
        tb = new(...);
        tb.run();
    end
endmodule : TB_top
    
```



static simulation structure

```

package TB_pkg;
class TB_env;
    ...
    virtual TB_hook hook;
    ...
endclass
    
```

dynamically constructed testbench object

Copyright © 2015-2023 Doulos. All Rights Reserved


65



## Constraints and Functional Coverage

### Class-Based SystemVerilog Verification

- What is SystemVerilog?
- SystemVerilog Classes
- Virtual Interfaces
- ➔ Constraints and Functional Coverage


66

### Constrained randomization

```

class Bus_trans;
  rand T_dir dir;
  rand T_addr addr;
  rand T_data data;
  constraint rom_area {
    dir == dir_Rd; addr <= 16'h7FFF;
  }
  ...
endclass : Bus_trans
        
```

constraints must be named

no semicolon!

semicolon!

System has ROM at low addresses

		dir	
		Rd	Wr
addr	FFFF	X	X
	8000	X	X
	7FFF	✓	X
	0000		

- But now there will be *no* access to high addresses!
- Solution: use an *implication constraint*


```

constraint low_adrs_is_ROM {
  (addr <= 16'h7FFF) -> (dir == dir_Rd);
}
        
```

		dir	
		Rd	Wr
addr	FFFF	✓	✓
	8000	✓	✓
	7FFF	✓	✓
	0000	✓	X

Copyright © 2015-2023 Doulos. All Rights Reserved
67

## Creating an Extended Class



```
class Bus_trans;
  rand T_dir dir;
  rand T_addr addr;
  rand T_data data;
```

General, re-usable

Better not to mix these together...

```
constraint low_adrs_is_ROM {
  (addr <= 16'h7FFF) -> (dir == dir_Rd);
}
```

Specific to the current DUT

- Don't *modify* the original class definition
- Instead, *extend* it:


```
class Mem_map_trans extends Bus_trans;
  constraint low_adrs_is_ROM {
    (addr <= 16'h7FFF) -> (dir == dir_Rd);
  }
  ...
```

Everything in the base class, plus...

Copyright © 2015-2023 Doulos. All Rights Reserved

68

## Inheriting Class Members



- What you write

```
Bus_trans
  addr: T_addr
  data: T_data ...
  new()
  copy(): Bus_trans
  psprint(): string
```

```
Mem_map_trans
  area: enum{ROM,RAM,IO}
  constraint...
```

Inherits,  
is-a...

- What you get

```
Mem_map_trans ... is a Bus_trans
  addr: T_addr
  data: T_data ...
  area: enum{ROM,RAM,IO}
  new()
  copy(): Bus_trans
  psprint(): string
  constraint...
```

- A *Mem\_map\_trans* object can be used anywhere a *Bus\_trans* object is appropriate

Copyright © 2015-2023 Doulos. All Rights Reserved

69

## Functional Coverage

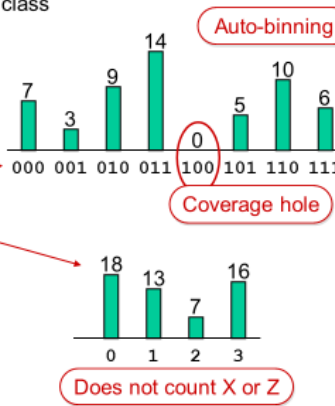
```

class Monitor;
  rand logic [2:0] opcode;
  rand logic [1:0] mode;

  covergroup cg;
    coverpoint opcode;
    coverpoint mode;
    option.per_instance = 1;
  endgroup

  function new;
    cg = new;
    ...
  task body;
    ... wait for transaction
    cg.sample();
    ...

```



Auto-binning

Coverage hole

Does not count X or Z

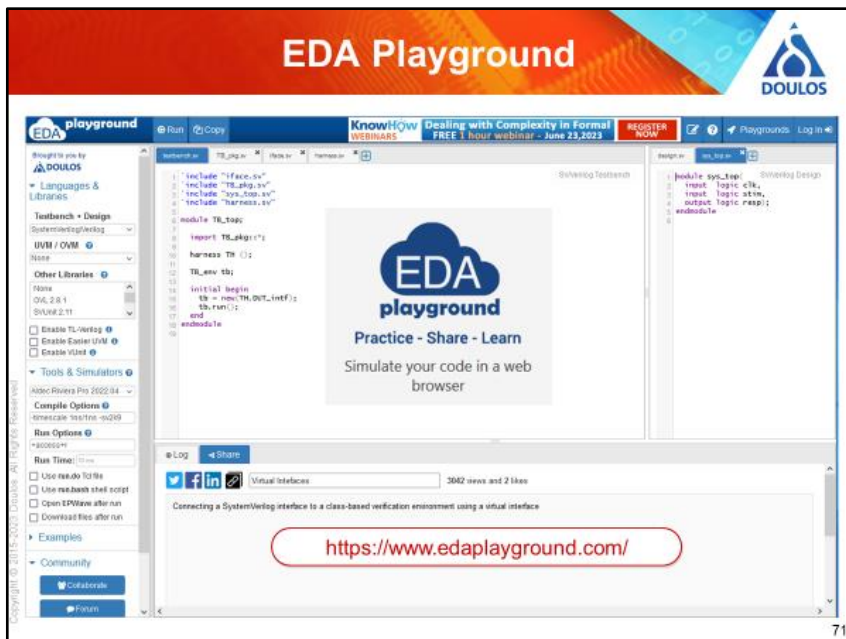
• Covergroup can be in a module, interface or class

Must instantiate covergroup in constructor!

Built-in method

70

## EDA Playground



https://www.edaplayground.com/

71


## Universal Verification Methodology (UVM)

### What is UVM?

### Universal Verification Methodology (UVM)


➔ What is UVM?

- Getting Started with UVM
- Should I use Formal instead?



72

### What is UVM?




- The Universal Verification Methodology for SystemVerilog
- Supports constrained random, coverage-driven verification
- An open-source (Apache 2.0) base class library
- An Accellera standard and the IEEE Standard 1800.2
- Supported by all major simulator vendors

Copyright © 2015-2023 Doulos. All Rights Reserved

73

## Why UVM?




- **Best practice**
  - Consistency, uniformity, don't reinvent the wheel, avoid pitfalls
- **Reuse**
  - Verification IP, verification environments, tests, people, knowhow

Copyright © 2015-2023 Doulos. All Rights Reserved

74

## Versions of UVM

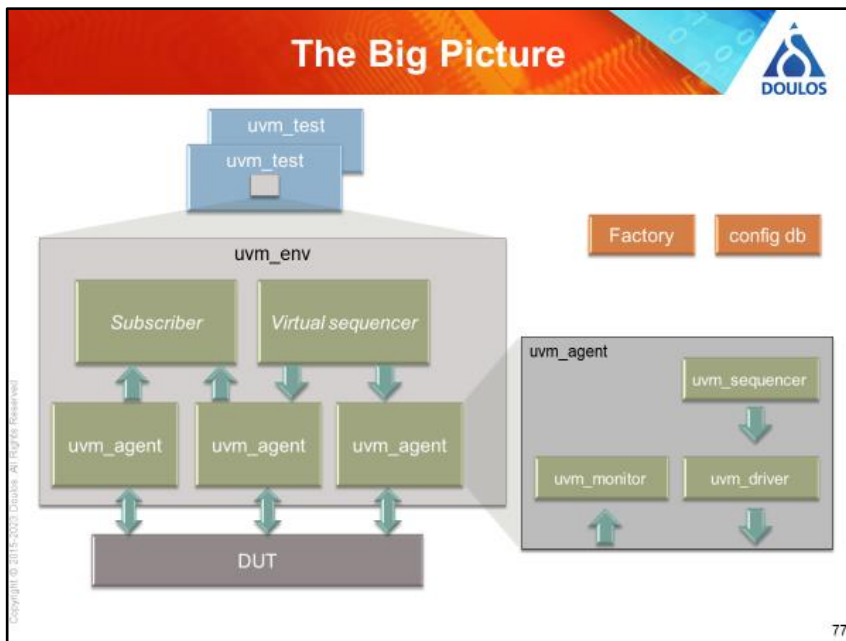
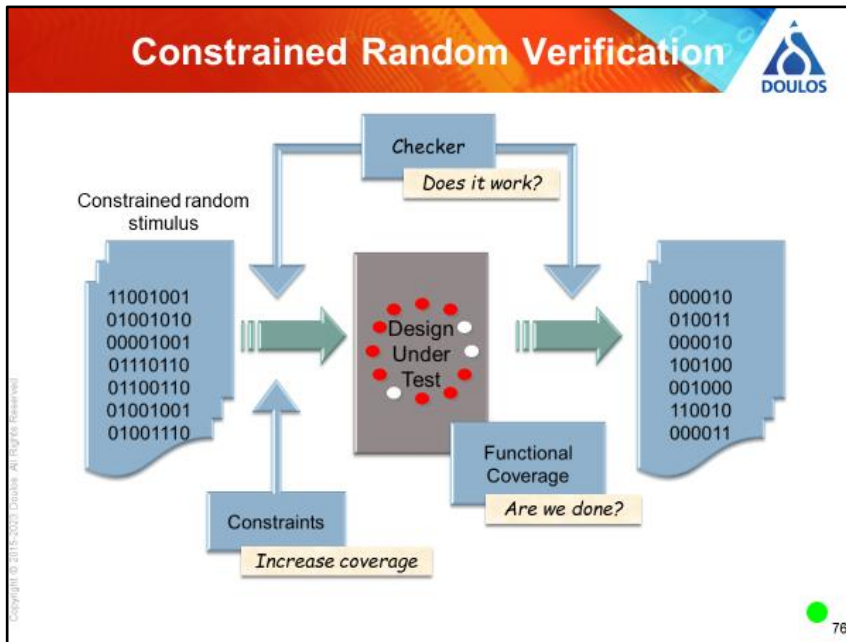


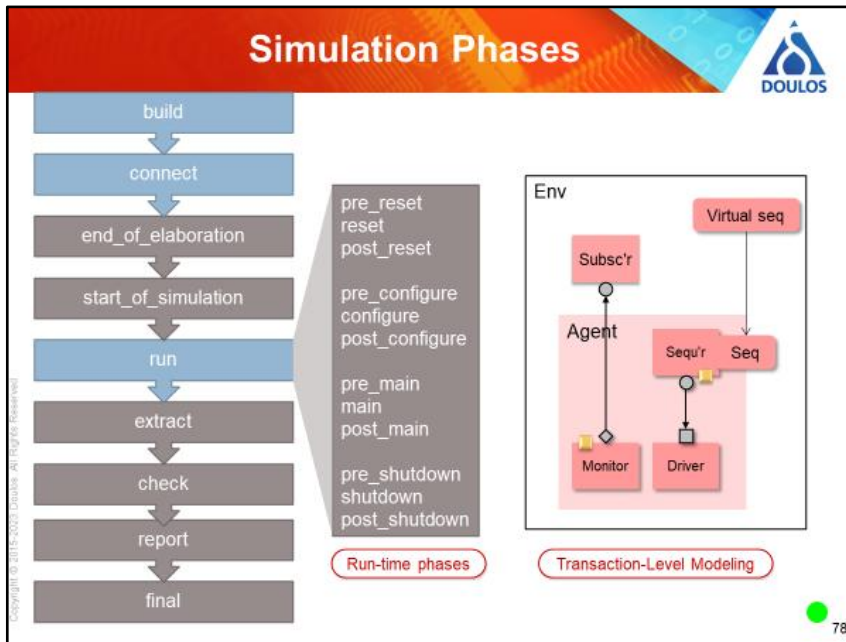
- UVM originally based on OVM 2.1.1
- UVM-1.2, June 2014
- IEEE Std 1800.2-2017 (Free from IEEE)
- UVM 2017-1.0 Reference Implementation
- IEEE Std 1800.2-2020 (Free from IEEE)
- UVM 2020-1.1 Reference Implementation

<http://www.accellera.org/downloads/standards/uvm/>  
<https://ieeexplore.ieee.org/document/9195920>

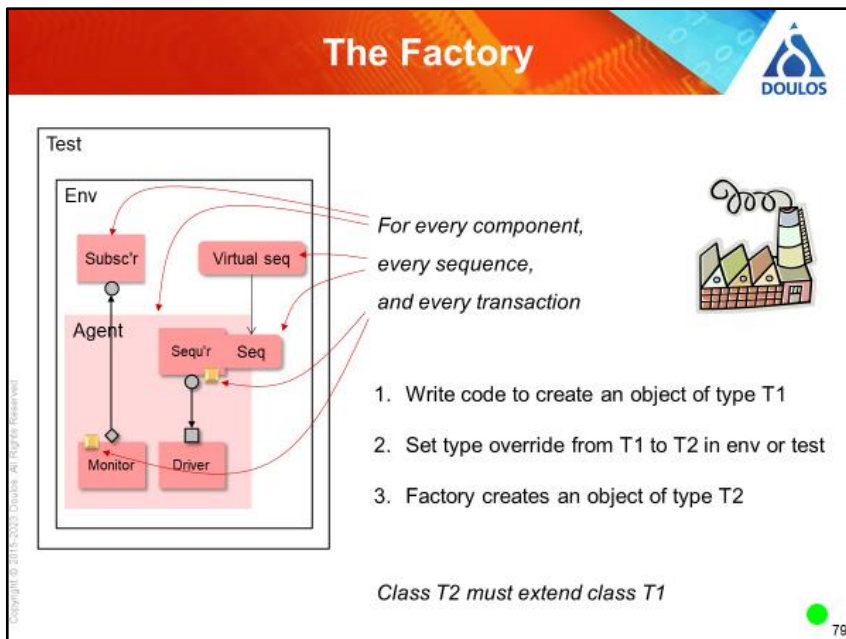
Copyright © 2015-2023 Doulos. All Rights Reserved

75





78




79

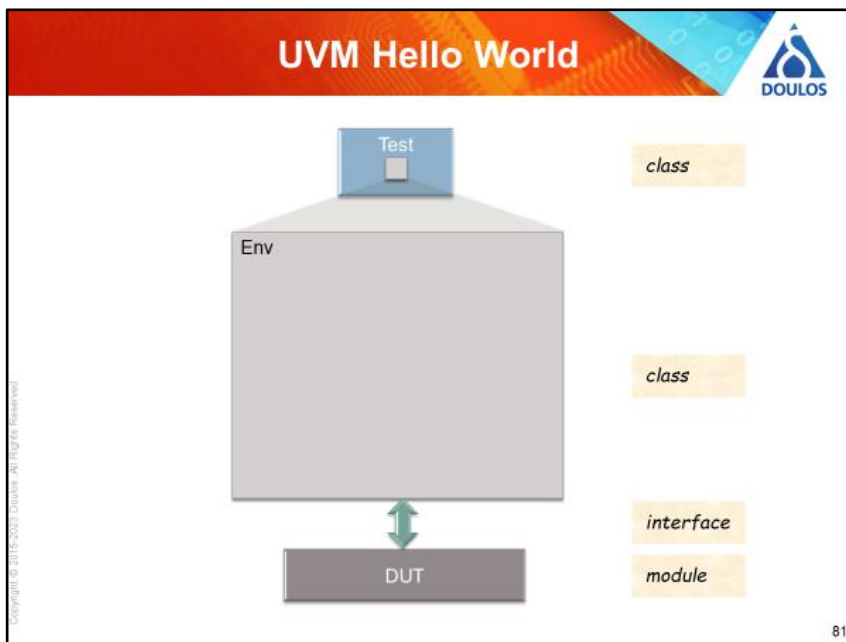
## UVM Hello World

### Getting Started with UVM

- UVM Hello World
- DUT Interface
- Sequencer-Driver Communication



80





## Interface and DUT



```
interface dut_if;  
endinterface
```

```
module dut(dut_if dif);  
endmodule
```

```
module top;  
...  
dut_if dut_if1 ();  
dut dut1 ( .dif(dut_if1) );  
...  
endmodule
```

Copyright © 2015-2023 Doulos. All Rights Reserved

82

## The Env




```
class my_env extends uvm_env;  
  `uvm_component_utils(my_env)  
  
  function new(string name, uvm_component parent);  
    super.new(name, parent);  
  endfunction  
  
endclass
```

Copyright © 2015-2023 Doulos. All Rights Reserved

83

## The Test (1)



```

class my_test extends uvm_test;

    `uvm_component_utils(my_test)

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction

    my_env m_env;

    function void build_phase(uvm_phase phase);
        m_env = my_env::type_id::create("m_env", this);
    endfunction


    task run_phase(uvm_phase phase);
        phase.raise_objection(this);
        ...
    endtask
    
```

UVM Factory

Copyright © 2015-2023 Doulos. All Rights Reserved

84

## The Test (2)



```

...

    m_env = my_env::type_id::create("m_env", this);
endfunction

task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    #10;
    `uvm_info("my_test", "Hello World", UVM_MEDIUM)

    phase.drop_objection(this);
endtask

endclass
    
```

UVM Objection

Copyright © 2015-2023 Doulos. All Rights Reserved

85

## Classes in a Package



```
`include "uvm_macros.svh"

package my_pkg;

    import uvm_pkg::*;

    class my_env extends uvm_env;
        `uvm_component_utils(my_env)
        ...
    endclass

    class my_test extends uvm_test;
        `uvm_component_utils(my_test)
        ...
    endclass

endpackage
```

Copyright © 2015-2023 by Doulos. All Rights Reserved

86

## Running the Test



```
interface dut_if;
endinterface
```

```
module dut(dut_if dif);
endmodule
```

```
module top;

    import uvm_pkg::*;
    import my_pkg::*;

    dut_if dut_if1 ();

    dut dut1 ( .dif(dut_if1) );


    initial
    begin
        run_test("my_test");
    end

endmodule
```

Copyright © 2015-2023 by Doulos. All Rights Reserved

87

## Hello World Source Code



```

interface dut_if;
endinterface

module dut(dut_if dif);
endmodule

module top;

import uvm_pkg::*;
import my_pkg::*;

dut_if dut_if1 ();
dut dut1 ( .dif(dut_if1) );

initial
begin
    run_test("my_test");
end

endmodule

package my_pkg;
import uvm_pkg::*;

class my_env extends uvm_env;
    `uvm_component_utils(my_env)

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction
endclass

class my_test extends uvm_test;
    `uvm_component_utils(my_test)

    my_env m_env;

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction


    function void build_phase(uvm_phase phase);
        m_env = my_env::type_id::create("m_env", this);
    endfunction

    task run_phase(uvm_phase phase);
        phase.raise_objection(this);
        #10;
        `uvm_info("", "Hello World", UVM_MEDIUM)
        phase.drop_objection(this);
    endtask
endclass
endpackage: my_pkg
    
```

Copyright © 2015-2023 Doulos. All Rights Reserved

88

## UVM Simulation Output



CDNS-UVM-1.2 (20.09-s003)  
 (C) 2007-2014 Mentor Graphics Corporation  
 (C) 2007-2014 Cadence Design Systems, Inc.  
 (C) 2006-2014 Synopsys, Inc.  
 (C) 2011-2013 Cypress Semiconductor Corp.  
 (C) 2013-2014 NVIDIA Corporation


...

UVM\_INFO @ 0: reporter [RNTST] Running test my\_test...  
**UVM\_INFO testbench.sv(58) @ 10: uvm\_test\_top [] Hello World**  
 UVM\_INFO /xcelium20.09/tools//methodology/UVM/CDNS-1.2/sv/src/base/uvm\_objection.svh(1271)  
 @ 10: reporter [TEST\_DONE] 'run' phase is ready to proceed to the 'extract' phase  
 UVM\_INFO /xcelium20.09/tools//methodology/UVM/CDNS-  
 1.2/sv/src/base/uvm\_report\_server.svh(847) @ 10: reporter [UVM/REPORT/SERVER]  
 --- UVM Report Summary ---

```

** Report counts by severity
UVM_INFO : 4
UVM_WARNING : 0
UVM_ERROR : 0
UVM_FATAL : 0
** Report counts by id
[] 1
[RNTST] 1
[TEST_DONE] 1
[UVM/RELNOTES] 1
        
```

Simulation complete via \$finish(1) at time 10 NS + 58 \*\*\*\*\*



<https://www.edaplayground.com/x/GjxC>


Copyright © 2015-2023 Doulos. All Rights Reserved

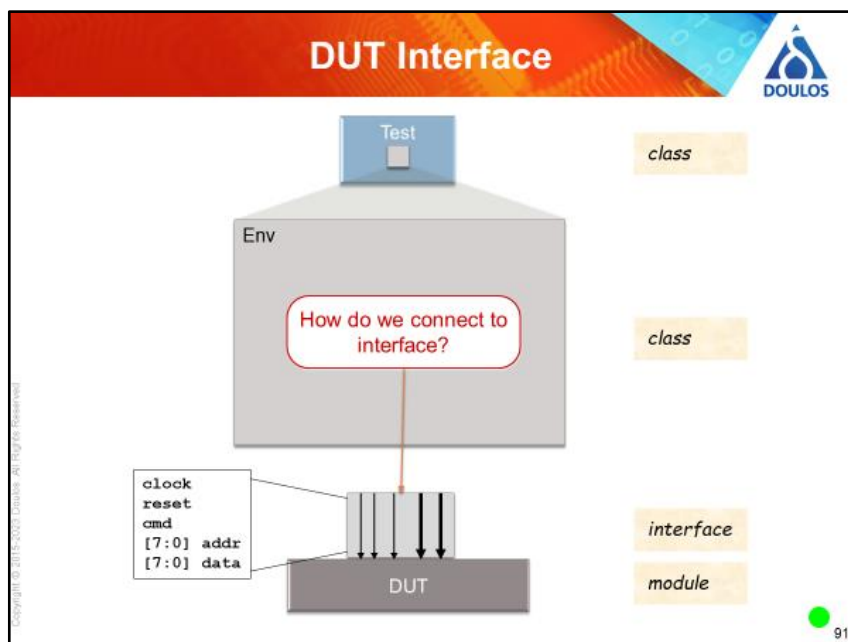
89

## DUT Interface

### Getting Started with UVM

- UVM Hello World
- ➔ • DUT Interface
- Sequencer-Driver Communication


90



## Interface and DUT

```

interface dut_if;
    logic clock, reset;
    logic cmd;
    logic [7:0] addr;
    logic [7:0] data;
endinterface
    
```

```

module top;
    import uvm_pkg::*;
    import my_pkg::*;

    dut_if dut_if1 ();
    dut dut1 ( .dif(dut_if1) );

    initial
    begin
        ...
        run_test("my_test");
    end
endmodule
    
```

```

module dut(dut_if dif);
    import uvm_pkg::*;

    always @(posedge dif.clock)
    begin
        `uvm_info("", $sformatf("DUT received cmd=%b, addr=%d, data=%d",
                                dif.cmd, dif.addr, dif.data), UVM_MEDIUM)
    end
endmodule
    
```

*Dummy implementation*

Copyright © 2015-2023 Doulos. All Rights Reserved

92

## Virtual Interface

Test

*class*

Env

*class*

Driver

*class*

virtual interface

*interface*

clock  
reset  
cmd  
[7:0] addr  
[7:0] data

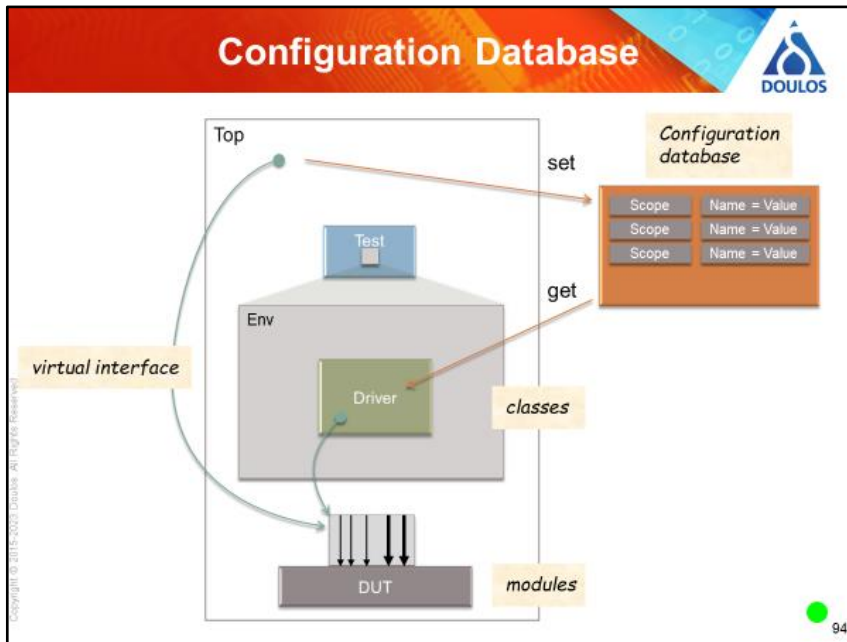
*module*

DUT

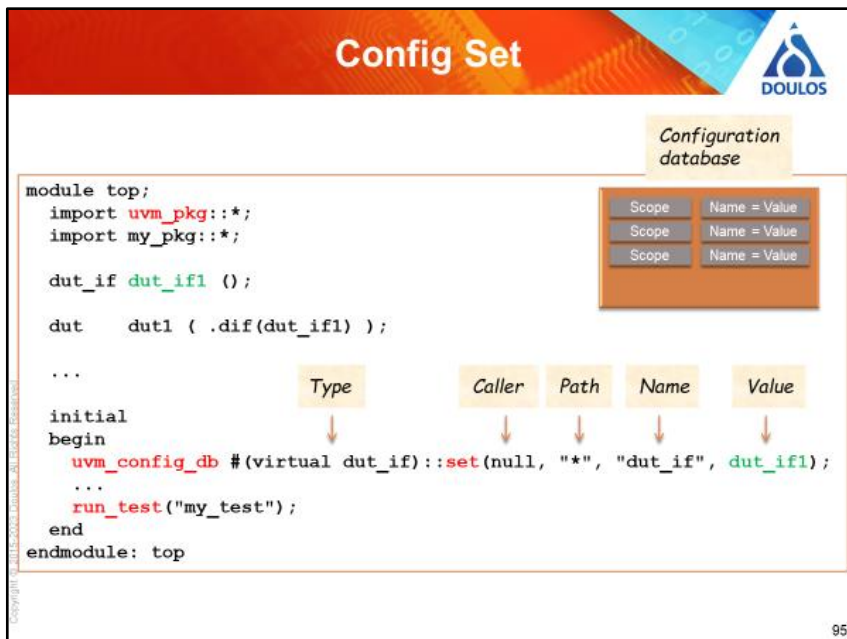
*module*

Copyright © 2015-2023 Doulos. All Rights Reserved

93




94



95

## Config Get



```

class my_driver extends uvm_driver;

  `uvm_component_utils(my_driver)

  virtual dut_if dut_vi;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);


    if( !
      Type      Caller  Path  Name  Value
      ↓        ↓       ↓   ↓   ↓   ↓
      uvm_config_db #(virtual dut_if)::get(this, "", "dut_if", dut_vi)
    )
      `uvm_error("", "uvm_config_db::get failed")

    endfunction
    ...
  endclass
    
```

Copyright © 2015-2023 Doulos. All Rights Reserved

96

## Pin Wiggling



```

class my_driver extends uvm_driver;

  `uvm_component_utils(my_driver)

  virtual dut_if dut_vi;

  ...

  task run_phase(uvm_phase phase);
    forever
    begin
      @(posedge dut_vi.clock);
      dut_vi.cmd <= $urandom;
      dut_vi.addr <= $urandom;
      dut_vi.data <= $urandom;
    end
  endtask

endclass
    
```

*Wiggle pins of DUT*

Copyright © 2015-2023 Doulos. All Rights Reserved


97



## Sequencer-Driver Communication

### Getting Started with UVM

- UVM Hello World
- DUT Interface
- ➔ • Sequencer-Driver Communication



98

### Sequence Item Class


```
class my_transaction extends uvm_sequence_item;

    rand bit cmd;
    rand bit [7:0] addr, data;

    function new (string name = "");
        super.new(name);
    endfunction


    `uvm_object_utils_begin(my_transaction)
        `uvm_field_int(cmd, UVM_DEFAULT)
        `uvm_field_int(addr, UVM_DEFAULT)
        `uvm_field_int(data, UVM_DEFAULT)
    `uvm_object_utils_end

endclass
```



99

## Sequence Class



```

class my_sequence extends uvm_sequence #(my_transaction);

    `uvm_object_utils(my_sequence)

    function new (string name = "");
        super.new(name);
    endfunction

    task body;
        repeat(8)
            begin
                `uvm_do(req)
            end
        endtask
    endclass
    
```


``uvm_do` creates and sends a randomized transaction to the driver

`req` inherited from `uvm_sequence`

Copyright © 2015-2023 Doulos. All Rights Reserved

100

## Sequence versus Sequencer



```

class my_sequence extends uvm_sequence #(my_transaction);
    ...
endclass

typedef uvm_sequencer #(my_transaction) my_sequencer;
    
```

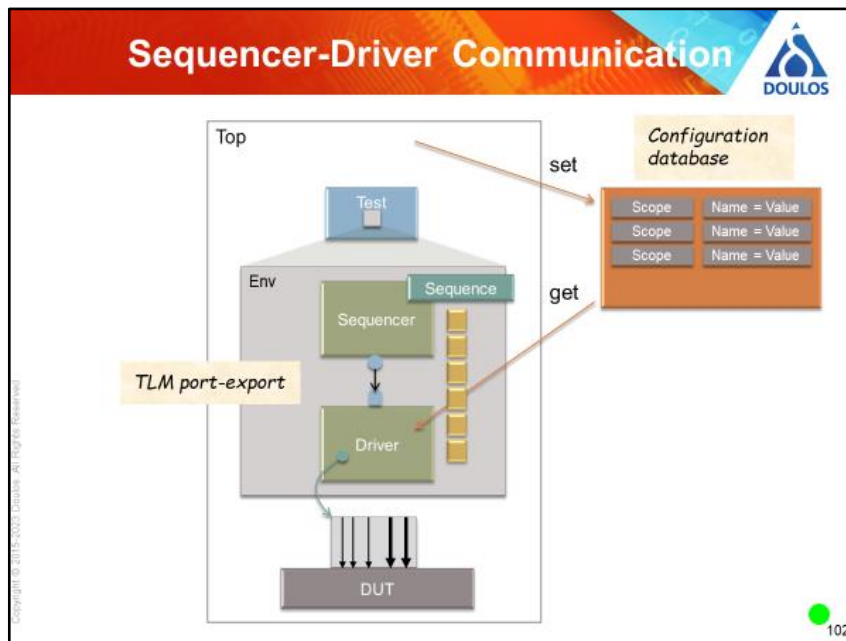
A sequence runs on a sequencer

```

uvm_sequence extends uvm_sequence_item extends uvm_object
uvm_sequencer extends uvm_component extends uvm_object
    
```

Copyright © 2015-2023 Doulos. All Rights Reserved

101



## Driver Run Phase

```

class my_driver extends uvm_driver #(my_transaction);
...
task run_phase(uvm_phase phase);
  forever
  begin
    seq_item_port.get(req);
    @(posedge dut_vi.clock);
    dut_vi.cmd = req.cmd;
    dut_vi.addr = req.addr;
    dut_vi.data = req.data;
  end
endtask
endclass

```


Pull sequence item from sequencer

Drive DUT through virtual interface

Copyright © 2015-2023 by Doulos. All Rights Reserved

103

## Sequencer-Driver Connection



```

class my_env extends uvm_env;

  `uvm_component_utils(my_env)

  my_sequencer m_seqr;
  my_driver    m_driv;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    m_seqr = my_sequencer::type_id::create("m_seqr", this);
    m_driv = my_driver    ::type_id::create("m_driv", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    m_driv.seq_item_port.connect( m_seqr.seq_item_export );
  endfunction


endclass

```

Copyright © 2015-2023 Doulos. All Rights Reserved

104

## Starting the Sequence



```

class my_test extends uvm_test;
  ...
  my_env m_env;
  ...

  task run_phase(uvm_phase phase);
    my_sequence seq;
    seq = my_sequence::type_id::create("seq");

    if( !seq.randomize() )
      `uvm_error("", "Randomize failed")

    seq.set_starting_phase(phase);
    seq.set_automatic_phase_objection(1);

    seq.start( m_env.m_seqr );
  endtask

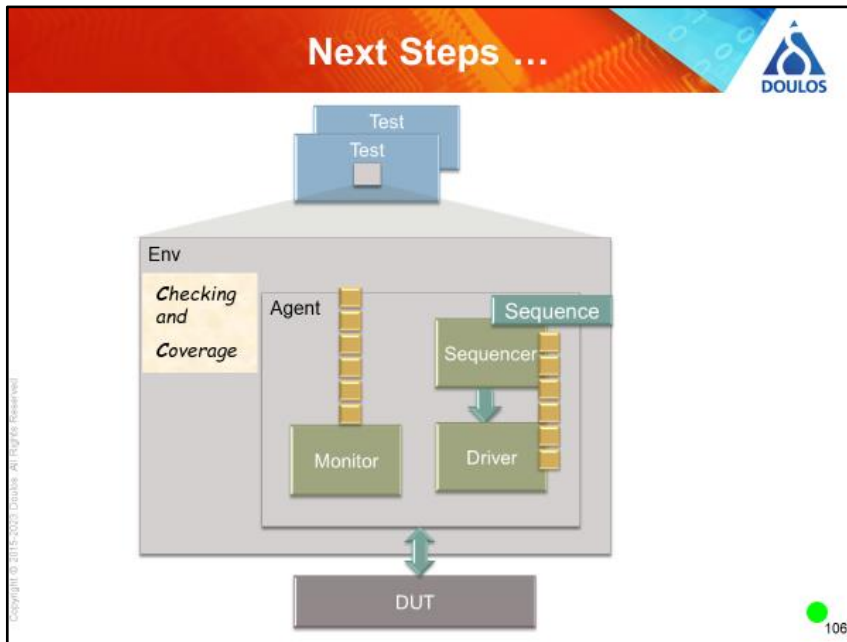
endclass

```

Run phase ends when all objections have been dropped


Copyright © 2015-2023 Doulos. All Rights Reserved

105



## Doulos – Easier UVM

- Coding guidelines – "One way to do it" Free and open
- Automatic code generator Apache 2.0 license



- Help individuals and project teams
  - learn UVM and avoid pitfalls
  - become productive with UVM (saves ~ 6 weeks)
  - use UVM consistently
- Reduce the burden of supporting UVM code


[https://www.doulos.com/knowhow/sysverilog/uvm/easier\\_uvm/](https://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm/)

107


## Formal Verification for Non-Specialists

### Learning to use Formal

### Formal Verification for Non-Specialists




- Learning to use Formal
- Writing Properties
- Tackling State Space
- Under-constraining versus Over-constraining
- Using Formal



108

### Why Formal Property Checking?



- Bug hunting
- Assurance, particularly of complex control logic
- Checking simple things like RTL linting, connectivity, coverage waivers
- Forces you to understand the spec
- Post-silicon debug

Copyright © 2015-2023 Doulos. All Rights Reserved

109

## Learning to Use Formal

- Formal is not hard to use!

Give it your RTL code  
Write some properties


Copyright © 2015-2023 Doulos. All Rights Reserved

110

### Writing Properties


## Formal Verification for Non-Specialists

- Learning to use Formal
- ➔ • Writing Properties
- Tackling State Space
- Under-constraining versus Over-constraining
- Using Formal



111

## RTL and Properties



**RTL**

```


module selAB (
  input logic clk,
  input logic QA, selA, QB, selB,
  output logic Q);

  always @(posedge clk)
  begin
    if (selA) Q <= QA;
    if (selB) Q <= QB;
  end


  check_selA: assert property (
    @(posedge clk) selA |>= Q == $past(QA) );
  check_selB: assert property (
    @(posedge clk) selB |>= Q == $past(QB) );

endmodule
                    
```

**Properties**

 112

## Learning to Use Formal



- Formal is not hard to use!

Give it your RTL code

Write some properties


Push the Go button

Wait for some results

Proof

Counter-example

Inconclusive

 113



## Log Window

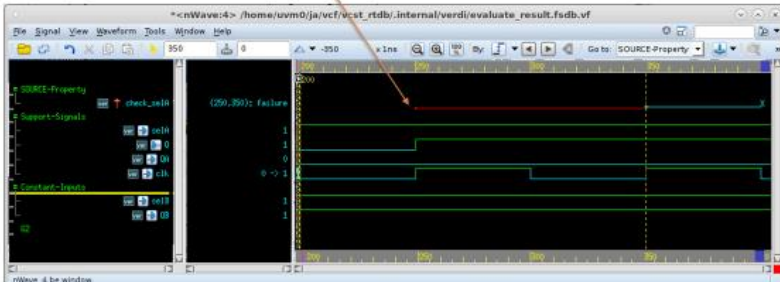
**SUMMARY**

Properties Considered	: 4	
assertions	: 2	
- proven	: 1 (50%)	←
- bounded_proven (user)	: 0 (0%)	←
- bounded_proven (auto)	: 0 (0%)	
- marked_proven	: 0 (0%)	
- cex	: 1 (50%)	←
- ar_cex	: 0 (0%)	
- undetermined	: 0 (0%)	
- unknown	: 0 (0%)	
- error	: 0 (0%)	
covers	: 2	
- unreachable	: 0 (0%)	
- bounded_unreachable (user)	: 0 (0%)	
- covered	: 2 (100%)	←
- ar_covered	: 0 (0%)	
- undetermined	: 0 (0%)	
- unknown	: 0 (0%)	
- error	: 0 (0%)	

114


## Counter-Example (CEX)

Property fails when SelA = SelB = 1 and QA != QB



115

## Learning to Use Formal



- Formal is not hard to use!


Give it your RTL code

Write some properties

Push the Go button

Wait for some results

Decide what to do next




Proof


Counter-example

Inconclusive

- The problem is understanding what formal can and cannot do

 116

## Formal Testbench




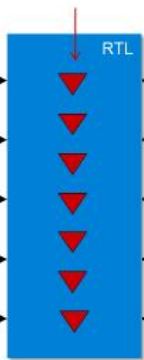
Input assumptions

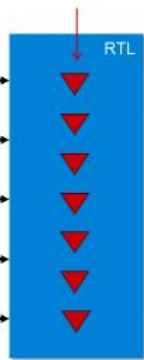
Local assertions

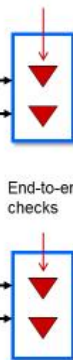
Local assertions

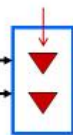
Protocol checks












 117

## Tackling State Space

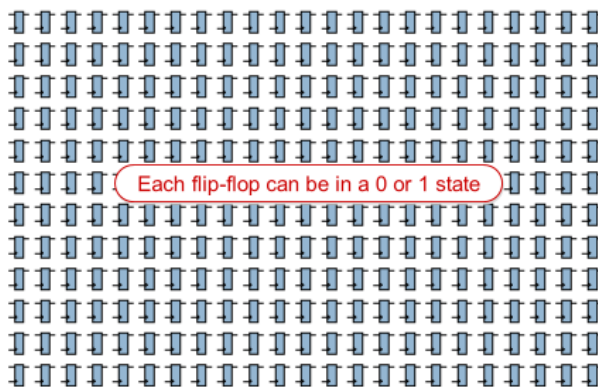
### Formal Verification for Non-Specialists

- Learning to use Formal
- Writing Properties
- ➔ • Tackling State Space
- Under-constraining versus Over-constraining
- Using Formal



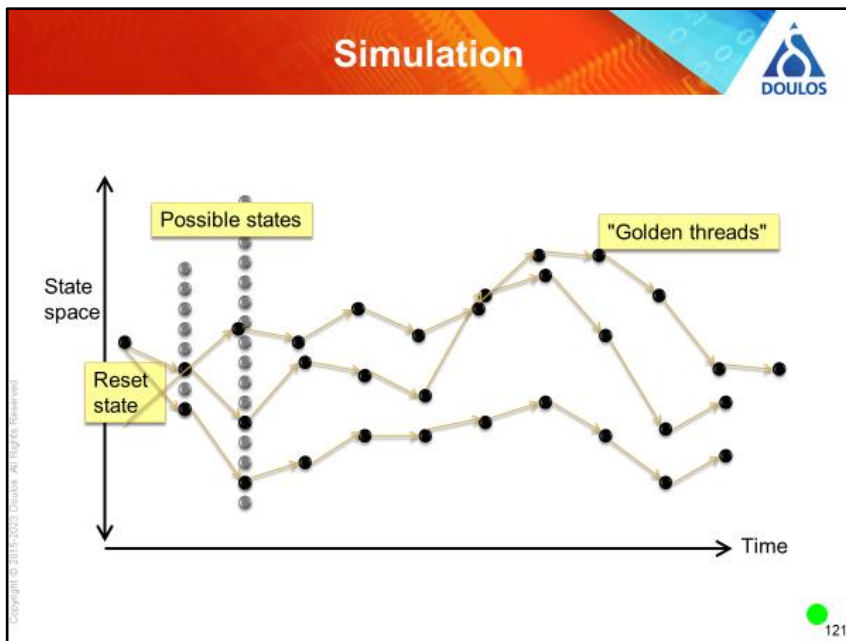
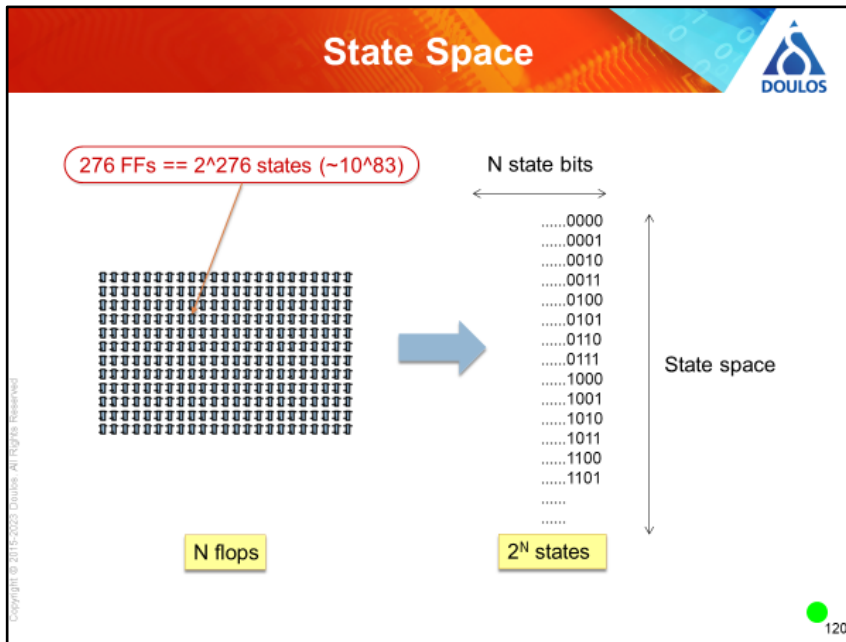
118

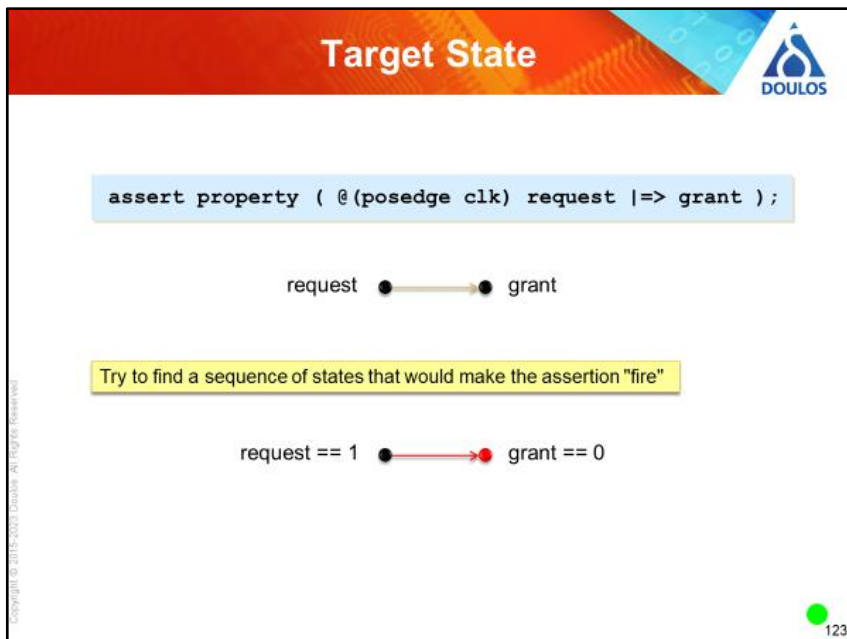
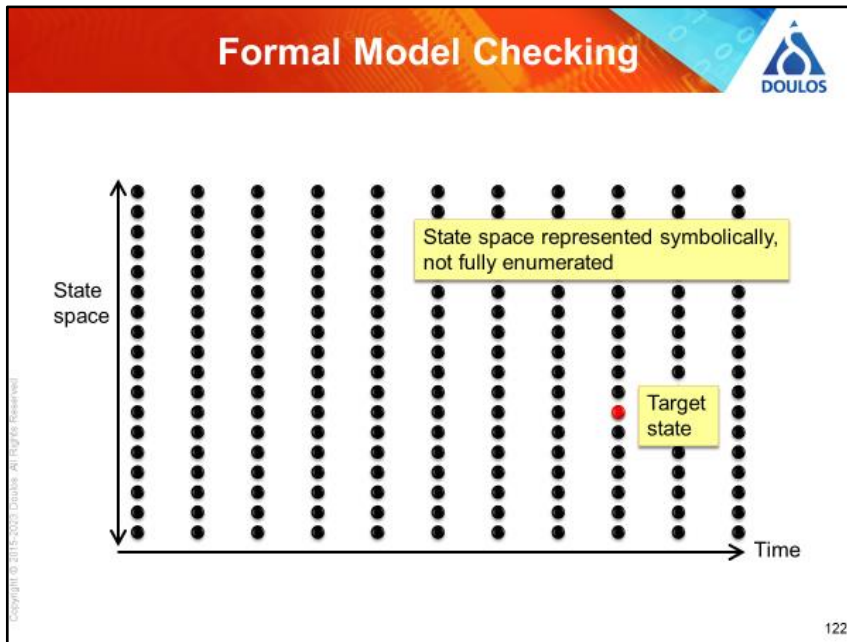
### State Space

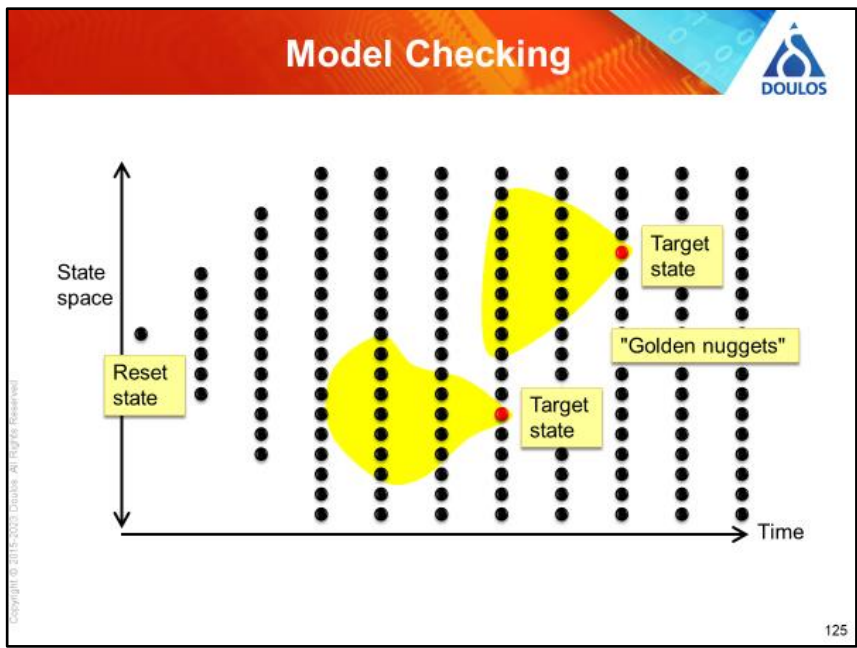
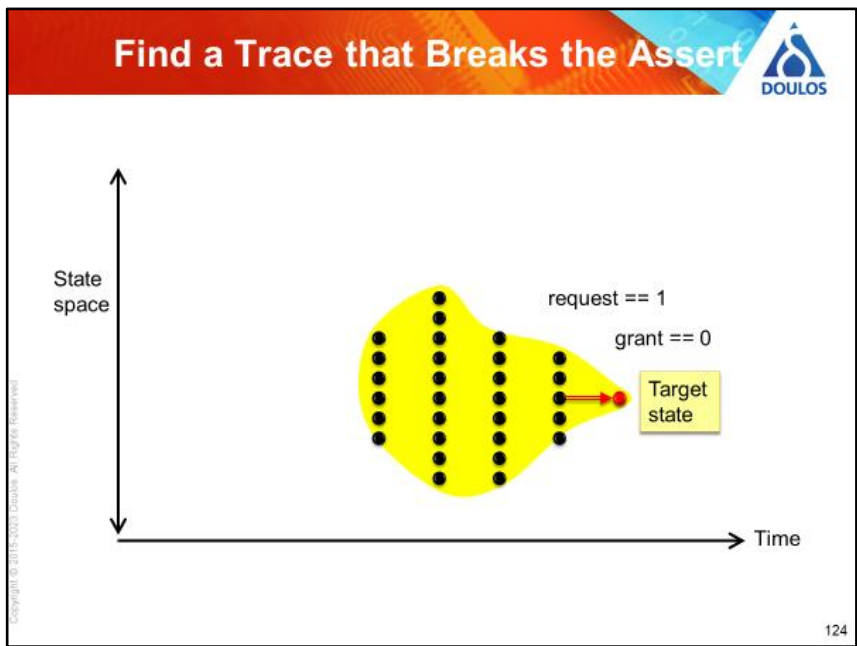


Copyright © 2015-2023 Doulos. All Rights Reserved

119







## Under-constraining versus Over-constraining

### Formal Verification for Non-Specialists

- Learning to use Formal
- Writing Properties
- Tackling State Space
- ➔ • Under-constraining versus Over-constraining
- Using Formal



126

### The Failing Example



```

Source Pane - visualize-5
Search the Source Code Pa... 1 of 2 COI
1 module selAB (
2   input logic clk,
3   input logic OA, selA, OB, selB,
4   output logic O;
5
6   always @(posedge clk)
7   begin
8     if (selA) O <= OA;
9     if (selB) O <= OB;
10  end
11
12  check_selA: assert property (
13    @(posedge clk) selA ==> O == $past(O));
14  check_selB: assert property (
15    @(posedge clk) selB ==> O == $past(O));
16
17 endmodule
  
```

Inputs are under-constrained

Selected: <embedded>::selAB.check\_selA Why at iteration 2\* for check\_selA (1 of 2) selAB.sv Visualize trace

127

## Under-constrained Inputs

selA	QA	selB	QB
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

```

always @(posedge clk)
begin
  if (selA) Q <= QA;
  if (selB) Q <= QB;
end
                    
```

Q
-
-
0
1
-
-
0
1
0
0
0
1
1
1
0
1

Illegal

Under-constrained inputs often give false negatives

Copyright © 2015-2023 Doulos. All Rights Reserved

128

## Input Assume Statement

```

module selAB (
  input logic clk,
  input logic QA, selA, QB, selB,
  output logic Q);

  always @(posedge clk)
  begin
    if (selA) Q <= QA;
    if (selB) Q <= QB;
  end

  check_selA:  assert property (
    @(posedge clk) selA | => Q == $past(QA) );

  check_selB:  assert property (
    @(posedge clk) selB | => Q == $past(QB) );

  assume_not_11: assume property (
    @(posedge clk) !(selA & selB) );


endmodule
                    
```

Copyright © 2015-2023 Doulos. All Rights Reserved

129



## Results



**SUMMARY**


```

Properties Considered      : 4
assertions                : 2
- proven                  : 2 (100%)
- bounded_proven (user)  : 0 (0%)
- bounded_proven (auto)  : 0 (0%)
- marked_proven          : 0 (0%)
- cex                    : 0 (0%)
- ar_cex                 : 0 (0%)
- undetermined           : 0 (0%)
- unknown               : 0 (0%)
- error                  : 0 (0%)
covers                    : 2
- unreachable            : 0 (0%)
- bounded_unreachable (user): 0 (0%)
- covered                : 2 (100%)
- ar_covered            : 0 (0%)
- undetermined          : 0 (0%)
- unknown               : 0 (0%)
- error                 : 0 (0%)
    
```

←

130

## Over-constrained Inputs



selA	QA	selB	QB
0	0	1	0
0	0	1	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	1	0	0
1	1	0	1

```

always @(posedge clk)
begin
  if (selA) Q <= QA;
  if (selB) Q <= QB;
end
    
```

Q
0
1
0
1
0
0
0
1
1

```

assume property (
  @(posedge clk) selA != selB );
    
```


Overconstrained inputs can give false positives

<del>0</del>	<del>0</del>	<del>0</del>	<del>0</del>
<del>0</del>	<del>0</del>	<del>1</del>	<del>0</del>
<del>0</del>	<del>1</del>	<del>1</del>	<del>0</del>
<del>0</del>	<del>1</del>	<del>1</del>	<del>1</del>

←

131

## Verifying Assumptions with Cover



```

always @(posedge clk)
begin
  if (selA) Q <= QA;
  if (selB) Q <= QB;
end

check_selA:  assert property (
              @(posedge clk) selA |=> Q == $past(QA) );

check_selB:  assert property (
              @(posedge clk) selB |=> Q == $past(QB) );


assume_not_11: assume property (
              @(posedge clk) selA != selB );

cover_00:    cover property (
              @(posedge clk) !selA & !selB );
    
```

Check input not over-constrained

132

## Results



SUMMARY	
Properties Considered	: 5
assertions	: 2
- proven	: 2 (100%)
- bounded_proven (user)	: 0 (0%)
- bounded_proven (auto)	: 0 (0%)
- marked_proven	: 0 (0%)
- cex	: 0 (0%)
- ar_cex	: 0 (0%)
- undetermined	: 0 (0%)
- unknown	: 0 (0%)
- error	: 0 (0%)
covers	: 3
- unreachable	: 1 (33.3333%)
- bounded_unreachable (user)	: 0 (0%)
- covered	: 2 (66.6667%)
- ar_covered	: 0 (0%)
- undetermined	: 0 (0%)
- unknown	: 0 (0%)
- error	: 0 (0%)

133

## Verifying Assumptions with Cover



```
always @(posedge clk)
begin
  if (selA) Q <= QA;
  if (selB) Q <= QB;
end

check_selA:  assert property (
              @(posedge clk) selA |=> Q == $past(QA) );

check_selB:  assert property (
              @(posedge clk) selB |=> Q == $past(QB) );

assume_not_11: assume property (
               @(posedge clk) !(selA & selB));

cover_00:    cover property (
             @(posedge clk) !selA & !selB );
```

Fix the assumption

134

## When things go wrong




- False negatives due to assertion bugs – debug CEX
- False negatives due to under-constrained inputs – debug CEX
- False positives due to over-constrained inputs – covers or simulation
- False positives due to insufficient assertions – assertion coverage
- False positives due to loose assertions – a challenge!

135

## Using Formal

### Formal Verification for Non-Specialists


- Learning to use Formal
- Writing Properties
- Tackling State Space
- Under-constraining versus Over-constraining
- ➔ • Using Formal



136

### Formal Use Model

- Use formal at block level, even before simulation
  - Find bugs before simulation
  - Forced to think about the spec
  - Properties carried around with the RTL
- Distinguish between local (simple) and end-to-end assertions
- Simple assertions are easy to write / understand / prove / debug
- End-to-end assertions are sometimes the most valuable




Copyright © 2015-2023 Doulos. All Rights Reserved

137


## Conclusions and Recommendations

### Conclusions and Recommendations



138

### Formal Complements Simulation

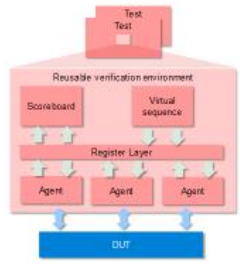
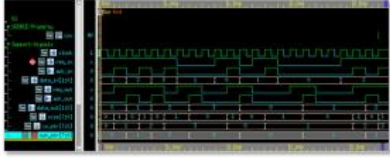


- Formal and simulation have different strengths and blind spots
- Formal will find bugs missed by simulation, and vice versa
- Formal encourages a different mindset from simulation

Copyright © 2015-2023 Doulos. All Rights Reserved

139

## UVM or Formal?

**Simulation (UVM)**

Of course!

But maybe not everything

**Formal**

Target pain points

Complement simulation

Include in verification planning

Copyright © 2015-2023 Doulos. All Rights Reserved.

140



Delivering KnowHow [www.doulos.com](http://www.doulos.com)

<b>SoC Design &amp; Verification</b>	» SystemVerilog » UVM » Formal » SystemC » TLM-2.0
<b>FPGA &amp; Hardware Design</b>	» VHDL » Verilog » SystemVerilog » Tcl » Xilinx » Intel FPGA (Altera)
<b>Embedded Software</b>	» Emb C/C++ » Emb Linux » Yocto » RTOS » Security » Arm
<b>Python &amp; Deep Learning</b>	 










141











[www.doulos.com](http://www.doulos.com)



