



PARADIGM[®]
WORKS



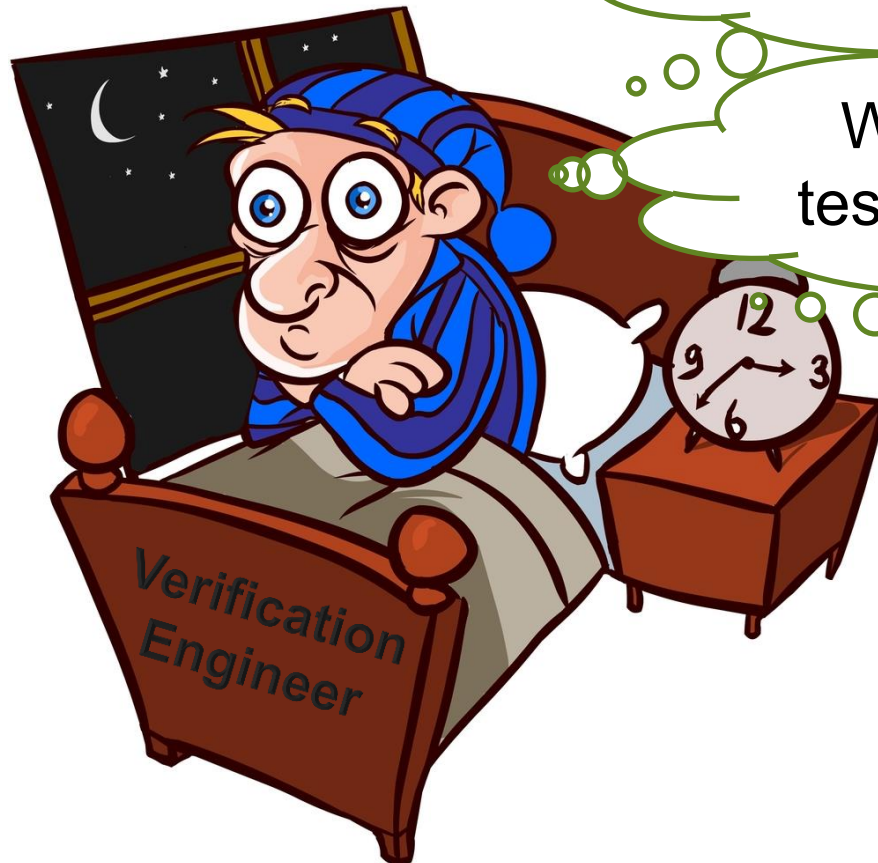
How Do You Know When Your Test Is Broken?

**Determining Test Quality through Dynamic Runtime
Monitoring of SystemVerilog Assertions**

Kelly D. Larson
Paradigm Works

September 23, 2014
Austin, TX USA

Is this you?



It's only 2 weeks 'til
tapeout.

We have over 10,000
tests in our regression...

I wonder if they all
work?...

How do you know your tests are doing what they should?

- A complex SoC can have well over 10,000 tests
- Making tests completely self-checking can be difficult

A test which 'passes' may simply mean that nothing "BAD" happened... And not necessarily that anything "GOOD" happened.



The cost of bad tests

- A 'broken' test steals valuable simulation cycles
- A 'broken' test is not testing what it should, and introduces a coverage hole
- Your coverage report may not be sufficient to expose these gaps



Solution

- We need a way to tie the pass/fail condition of an individual test to the specific conditions or goals of the test
- This needs to be a scalable solution
- This needs to work with both constrained random tests, as well as processor-centric style directed testing



Assertion Monitor!

Specific Test Requirements

- My test was supposed to hit a specific coverage point or fail. Did it?
- I know my test was supposed to make condition 'X' happen exactly five times or fail. Did it?
- Because of the way that I wrote my test, I should never see 'Y' happen, and if it does I want the test to fail even though 'Y' itself is not illegal. Will it?

Assertion Monitor

- Most standard use of SystemVerilog assertions is to target DESIGN QUALITY
- Our Assertion Monitor solution will target TEST QUALITY

Design
Quality

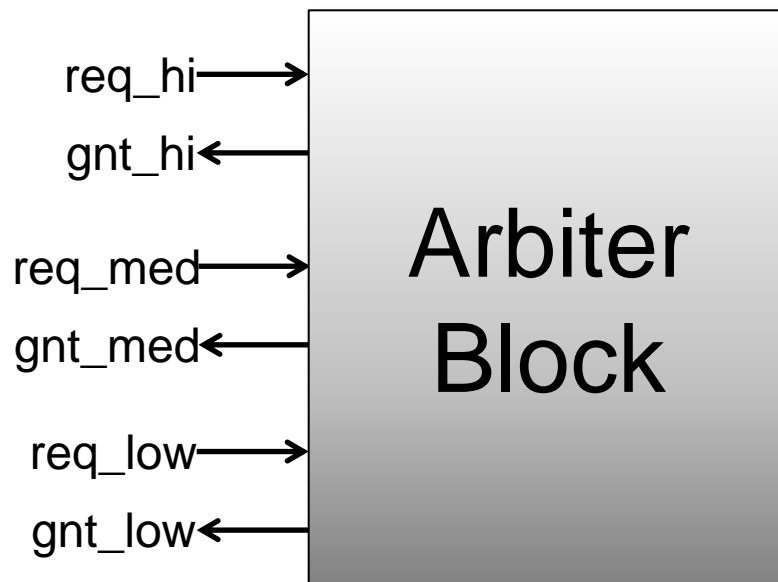


Test Quality

What about Assertion Coverage?

- Using a dynamic assertion monitor is similar to analyzing assertion coverage reports, except:
 - Focus is on individual tests, not overall results
 - Done while simulation is running
 - Can fail test immediately upon detecting a problem with the test
 - More flexible, can fail for condition hit or not hit, or hit within defined ranges

Example Use Case: Arbiter



```
check_hi: assert property(@(posedge clk) disable iff (reset)  
    req_hi |==> gnt_hi);
```

```
check_med: assert property(@(posedge clk) disable iff (reset)  
    (req_med & !req_hi) |==> gnt_med);
```

```
check_low: assert property(@(posedge clk) disable iff (reset)  
    (req_low & !req_med & !req_hi) |==> gnt_low);
```

Make it interesting...

```
check_arb: cover property(@(posedge clk) disable iff (reset)
                        (req_hi & req_med & req_low));
```

- The assertions in this example will catch illegal activity, but they won't actually insure that any 'arbitration' actually occurred
- In this case, we'll need add another cover point to observe an interesting condition

Plusarg directives

- What we'd like now is to be able to run a VCS simulation with an additional argument which requires our interesting condition to occur in order for the test to pass

```
>simv +RequireAssert=check_arb
```

- How about a test where I know a particular condition should not occur?

```
>simv +ProhibitAssert=check_hi
```

Assertion Monitor Plusargs

Type	# Arg	Example & Description	When Checked
Require	0	+RequireAssert=myassert. Assertion must fire at least once during the test.	End of test
	1	+RequireAssert=myassert:x, Assertion must fire at least 'x' times during the test.	End of test
	2	+RequireAssert=myassert:x:y, Assertion must fire in the range greater than or equal to 'x', and less than or equal to 'y' times.	During (too many), End (too few)
Prohibit	0	+ProhibitAssert=myassert, Assertion must never fire during the simulation.	During Test
	1	+ProhibitAssert=myassert:x, Assertion must not fire 'x' or more times (less is OK).	During Test
	2	+ProhibitAssert=myassert:x:y, Assertions cannot fire in the range of [x:y] inclusive (less or more is OK).	End of test

What's in a name?

```
>simv +RequireAssert=my_assert
```

- Ideally we only want to have to specify the name of the assertion
 - Hierarchical paths are messy, and prone to changes

```
>simv +RequireAssert=my_module1.my_assert
```

- If assertion is instantiated multiple times, we'll need to add just enough path to disambiguate
 - Monitor should alert us if we need more

Multiple arguments

```
>simv +RequireAssert=my_assert1:2,my_assert2:1:5 \  
      +ProhibitAssert=path.my_assert3,another_path.my_assert3 \  
      +RequireAssert=yet_another_assert:8 \  
      +RequireAssert=big.path.and_another
```

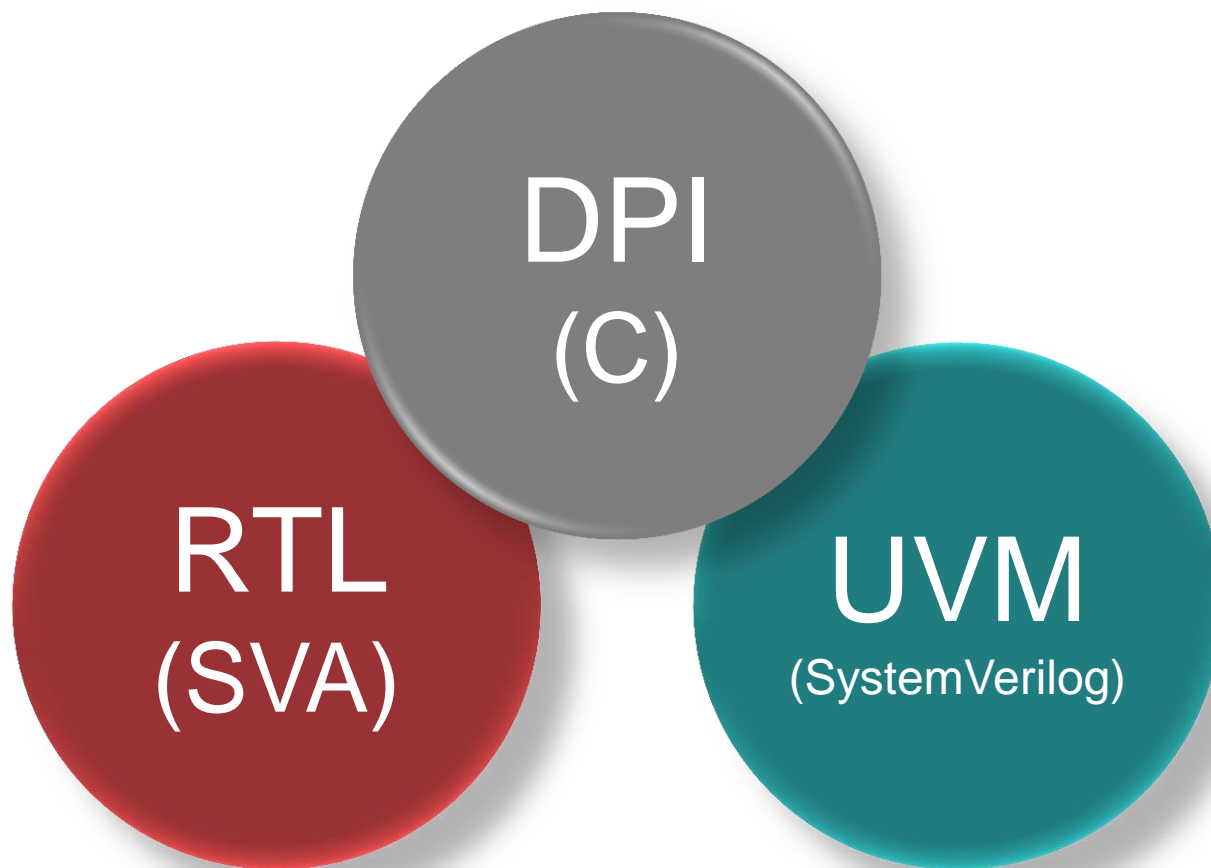
- We should be able to specify multiple assertions to monitor
 - Either multiple plusargs, comma delimited, or both



Looks good...
let's build it!

Assertion Monitor Components

- Our assertion Monitor has three main components:



RTL Component



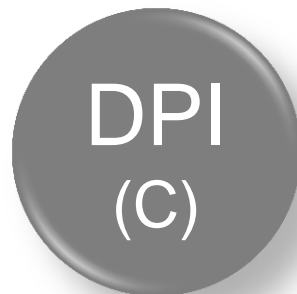
- RTL is instrumented with SVA assertions and coverpoints
 - Ideally we can make use of existing assertions written for design quality
- Assertion monitor treats assertions and coverpoints as the same
 - From test quality perspective we don't really 'care' if the assertions passes or fails, only that the condition was tested

UVM Component



- Before the test begins, parse command line directives and call DPI routine to instrument assertion tracking
- At the end of the test, do final check for proper behavior
- Provide utility functions that will allow the assertion monitor DPI routines to report UVM errors and warnings

C DPI Component



- Provide a data structure to store runtime information about monitored assertions
- Provide the mechanism to attach a callback routine to monitored assertions
- Provide the callback routine which will be run every time a monitored assertion or cover point successfully passes.
- Provide an 'end of test' routine which will do a final check

SystemVerilog Assertion API

Our assertion monitor makes use of two key features of the Assertion API:

- 1) The ability to iterate through a design to find specific assertions
- 2) The ability to attach our own callback (subroutine) to an assertion which will get called whenever the assertion (or cover point) passes successfully


Iterating through RTL Assertions

```
itr = vpi_iterate(vpiAssertion, NULL);  
while (assertion = vpi_scan(itr)) {  
    /* process assertion */  
}
```

- Assertion API allows us to easily iterate through handles of all of the assertions and coverpoints in the design
- Handle gives us access to the hierarchical path, and allows us to attach a callback

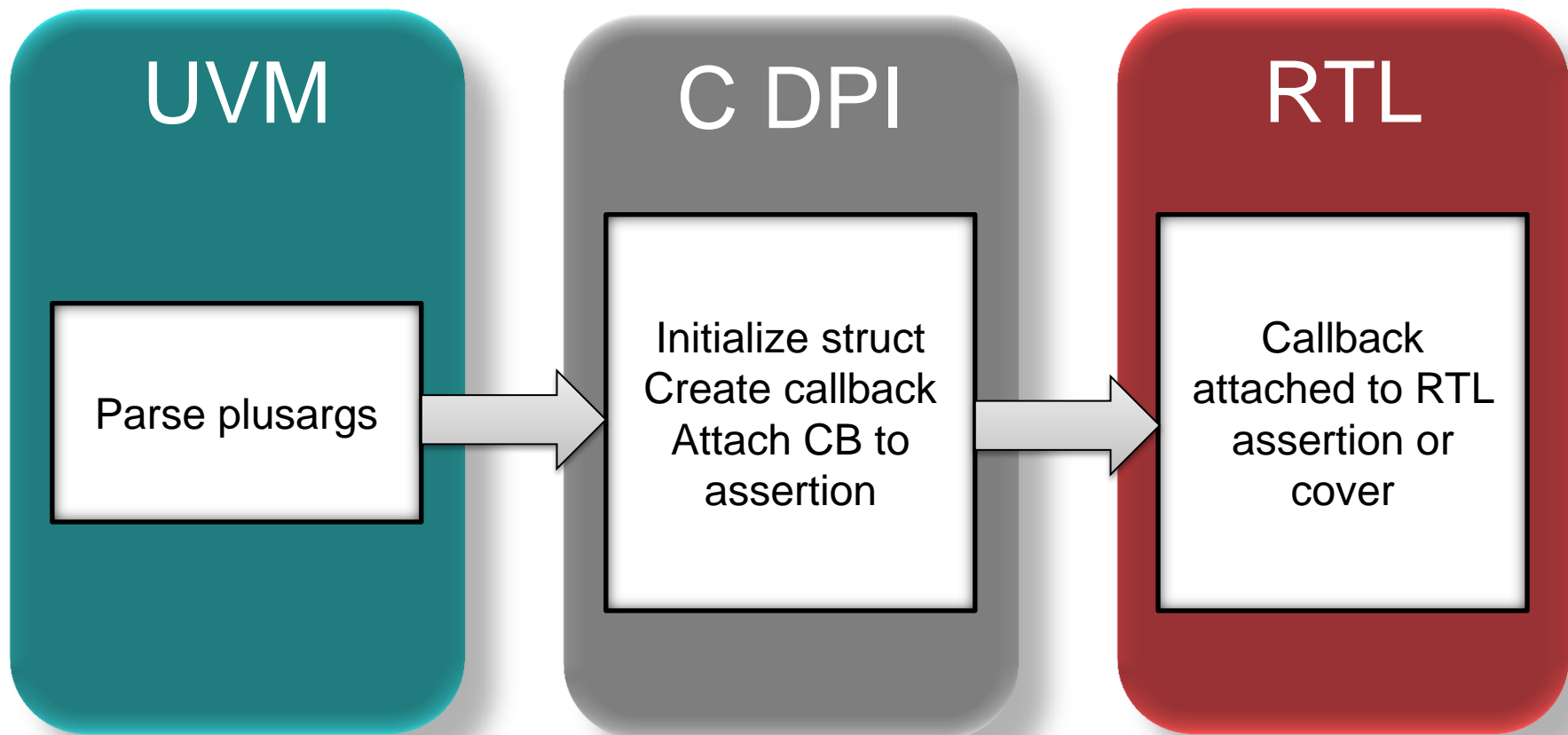
Registering a Callback

```
vpiHandle vpi_register_assertion_cb(  
    vpiHandle assertion, /* handle to assertion */  
    PLI_INT32 reason,    /* reason for which callbacks needed */  
    vpi_assertion_callback_func *cb_rtn,  
    PLI_BYTE8 *user_data /* user data to be supplied to cb */  
);
```

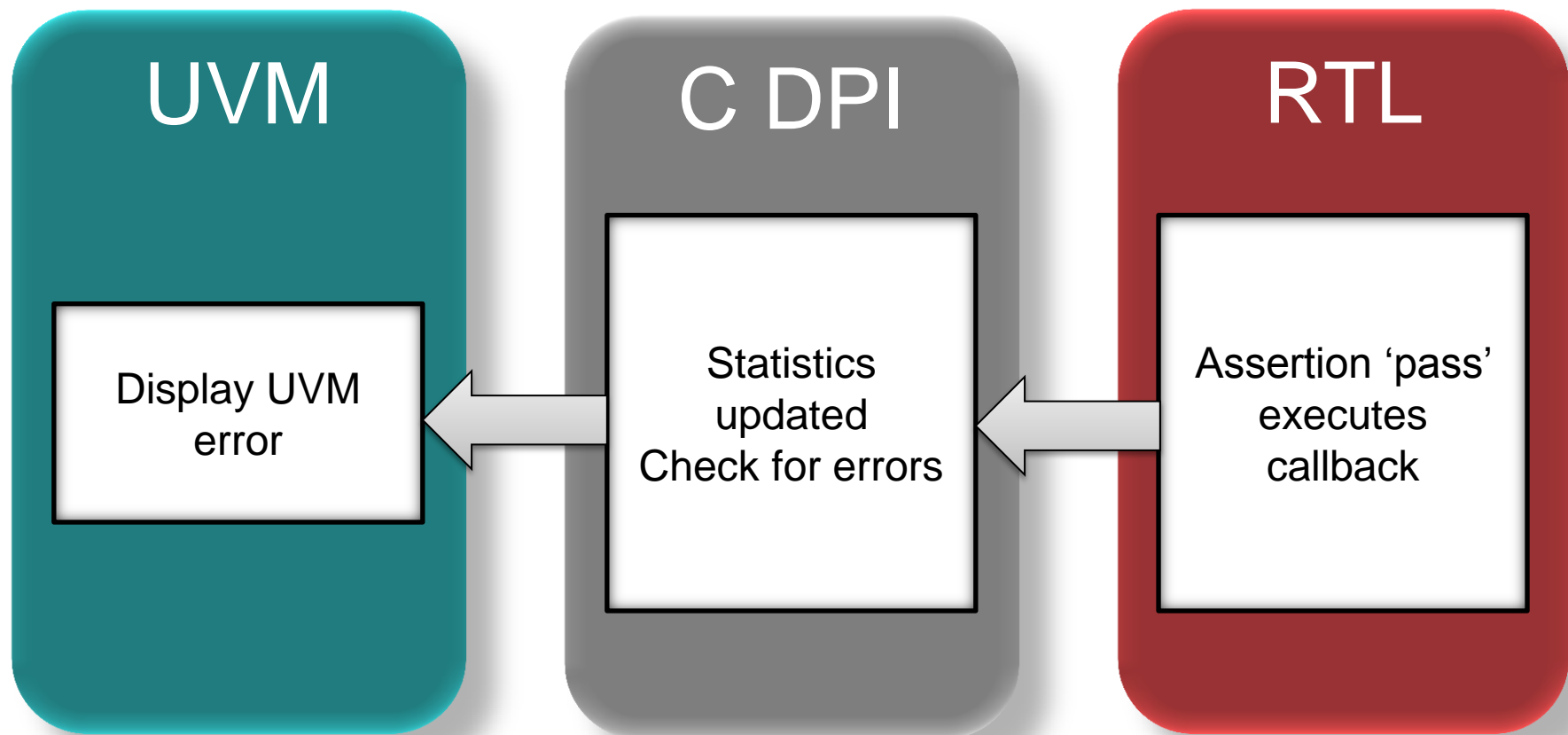


```
typedef PLI_INT32 (vpi_assertion_callback_func) (  
    PLI_INT32 reason,          /* callback reason */  
    p_vpi_time cb_time,       /* callback time */  
    vpiHandle assertion,      /* handle to assertion */  
    p_vpi_attempt_info info,  /* attempt related information */  
    PLI_BYTE8 *user_data      /* registered user data */  
);
```

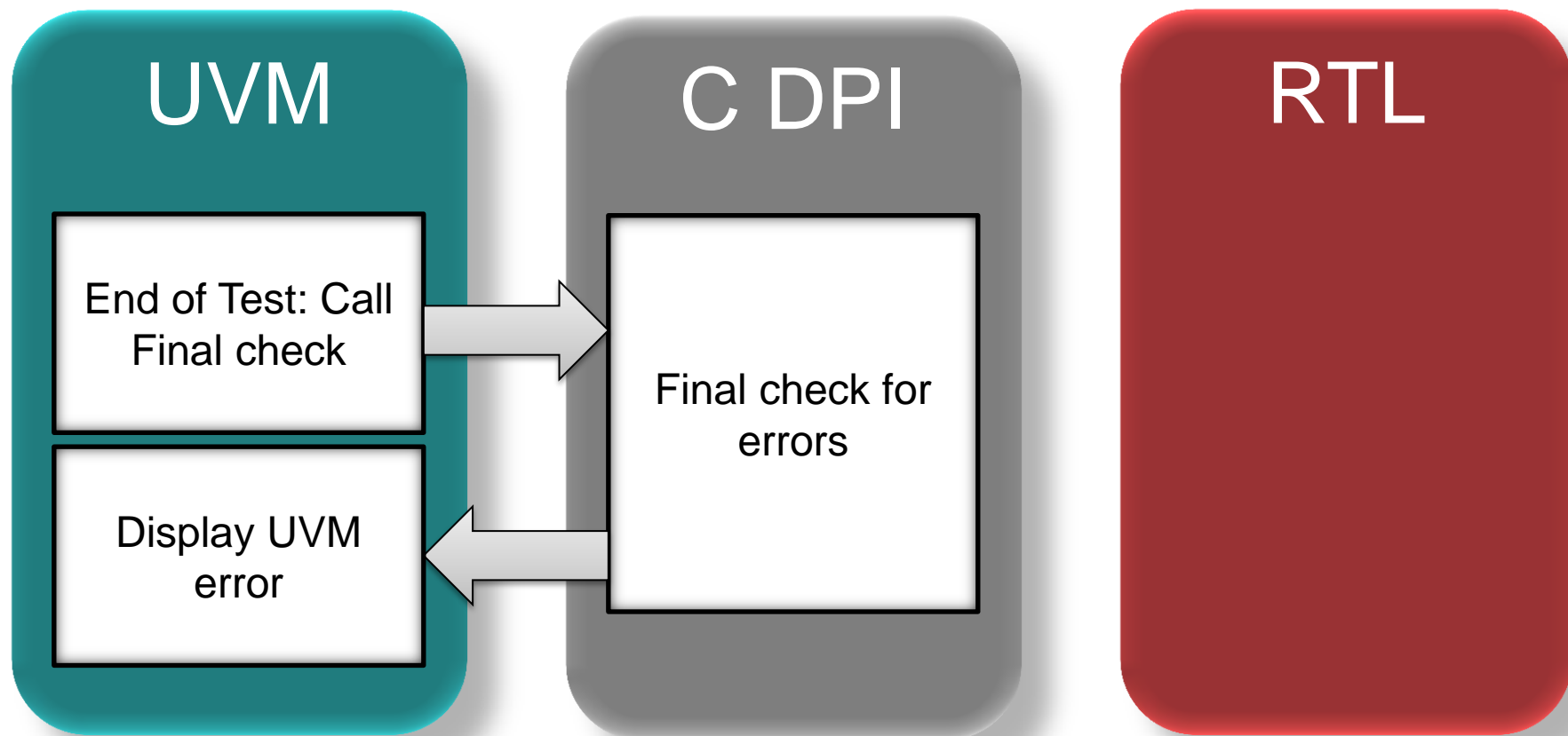
Flow: Before Test



Flow: During Test



Flow: After Test



Don't like plusargs?

- Assertions can be monitored by calling DPI routines directly from the testbench
- Can be called from within random testcases to provide dynamic feedback to help guide the progression of the testcase itself



Constrained Random Testing

```
class bus_seq extends uvm_sequence #(bus_txn);  
  <...>  
  virtual task body();  
    ahandle = register_assert("upper.overflow_detect",-1,-1,1);  
    while (successes < 3) begin  
      `uvm_do(tr);  
      successes = num_assert_successes(ahandle);  
    end  
  endtask: body  
endclass: bus_seq
```

- In this example, assertion is registered with the DPI directly from the UVM sequence
- Sequence actively monitors the assertion until 3 successful passes are detected

Other fun tricks...

- Assertion API can also be used to enable and disable assertions on the fly
 - Can be used during “error injection” type tests
 - Can be used during particular power modes
 - Can be used for ‘temporary’ work-arounds

However, use with...



...one last caveat

- To get accurate results with VCS you must run the simulation with the following option:

```
-assert cbSuccessOnlyNonVacuous
```

- Synopsys has said that this will become default behavior on a future release

Summary

- It's more important than ever to make sure every simulation cycle is well spent
- 'Broken' tests not only waste cycles, but add risk by exposing unwanted coverage holes
- SystemVerilog has built-in facilities that allow us to dynamically track assertions and coverpoints during the test
- Dynamically tracking coverage helps insure that a test continues to do what it's supposed to do throughout the project

Thank You

kelly.larson@paradigm-works.com