Verification as a Mature Discipline:

Using the

Verification Methodology Manual for SystemVerilog

Jonathan Bromley

Doulos Ltd, Ringwood, UK

Aims

- Road map
  - Covers highlights and interesting features from the VMM
  - Many additional concepts and details in the book itself
- Key ideas
  - Explore overall structure, motivation, techniques
- Practical challenges
  - Verification methodology must operate in the real world
  - Uses some advanced SystemVerilog features

Agenda

- **Motivation and background**
- Introduction to VMM test environment
- SystemVerilog classes
- Modeling test data
- Modeling the test environment
- Self-checking, assertions and coverage
- Creating testcases
- System-level verification (preview)
- Conclusions

Motivation

- All verification environments should follow industry best-practice and shared conventions
  - for flexibility, interoperability, re-use, verification confidence
- Verification is complicated and difficult
  - needs specific guidelines and ready-to-use infrastructure
- Adopt proven prior art wherever possible

Meeting the challenges

- Common infrastructure must support diverse applications
  - Standard design patterns, agreed approaches
  - Class inheritance and polymorphism permits application-specific extensions of standard library

- Verification practice needs a scientific basis
  - Proven benefits of random stimulus generation
  - Functional coverage for measurable verification productivity

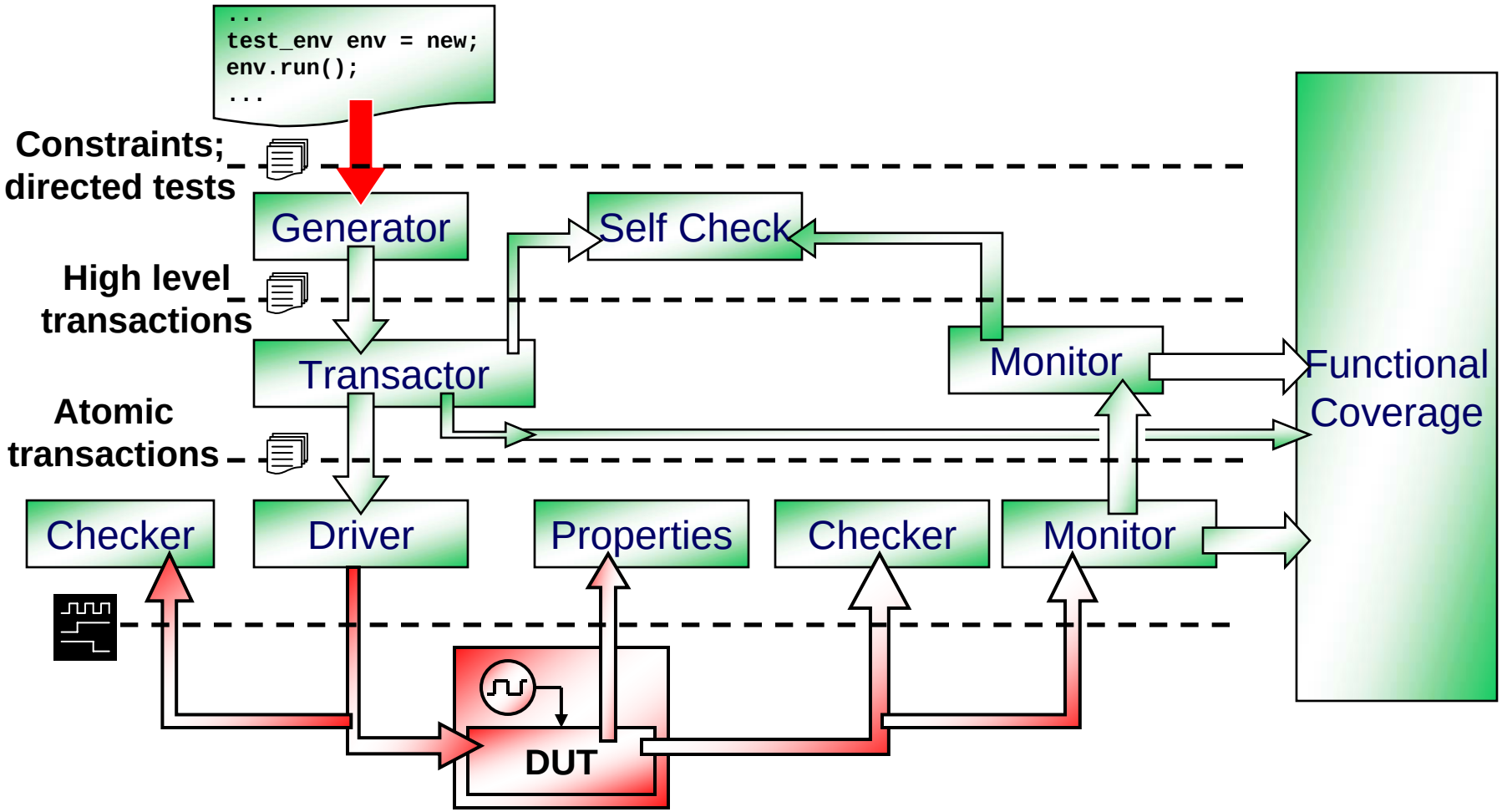
VMM background

- Jointly created by Synopsys and ARM
- Incorporates...
  - ARM experience in system level verification
  - Synopsys experience with advanced RTL verification
- Builds on existing Synopsys reference verification methodology

Agenda

- Motivation and background
- **Introduction to VMM test environment**
- SystemVerilog classes
- Modeling test data
- Modeling the test environment
- Self-checking, assertions and coverage
- Creating testcases
- System-level verification (preview)
- Conclusions

Overall architecture (*preview*)



Support infrastructure

- VMM Standard Library provides
  - Message logging service
  - Event notification service
  - Base classes for
    transaction, transactor, channel, environment
- VMM-compliant environments *must* use these
- Highly user-configurable through class inheritance

Logging

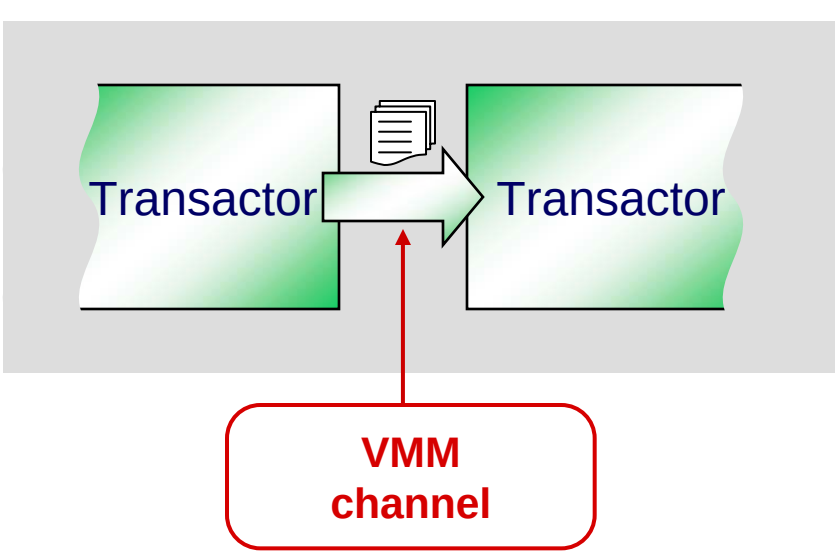
- Verification environment must report interesting activity, but we need…
  - Consistency — *post-processing and interpretation*
  - Localisation — *identification and control of source*
  - Configurability — *control of message severity and type*
  - Control — *effect on simulation behaviour*
- NEVER use `$display` for message logging!

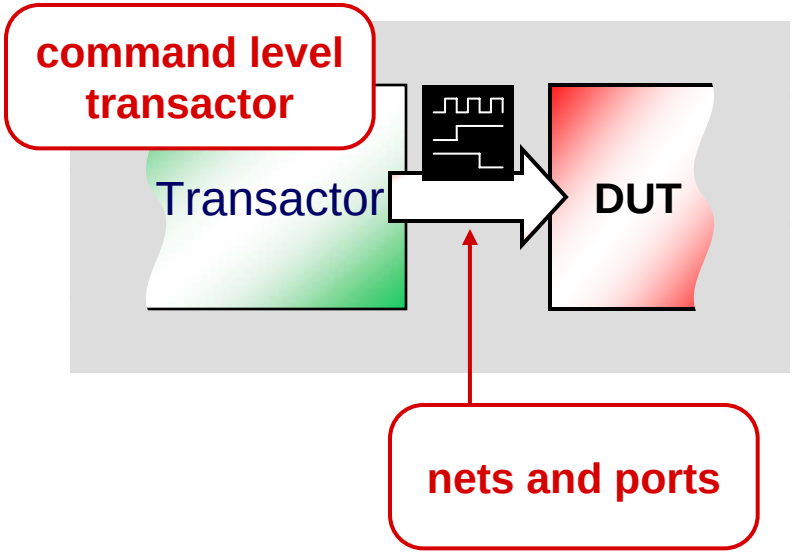
Notification

- Like events, but much more flexible
- Used by transactors and channels to mark activity
- Every transactor keeps a reference to the notification service instance that it uses

Transactors and channels

- Transaction objects are produced by a *transactor*





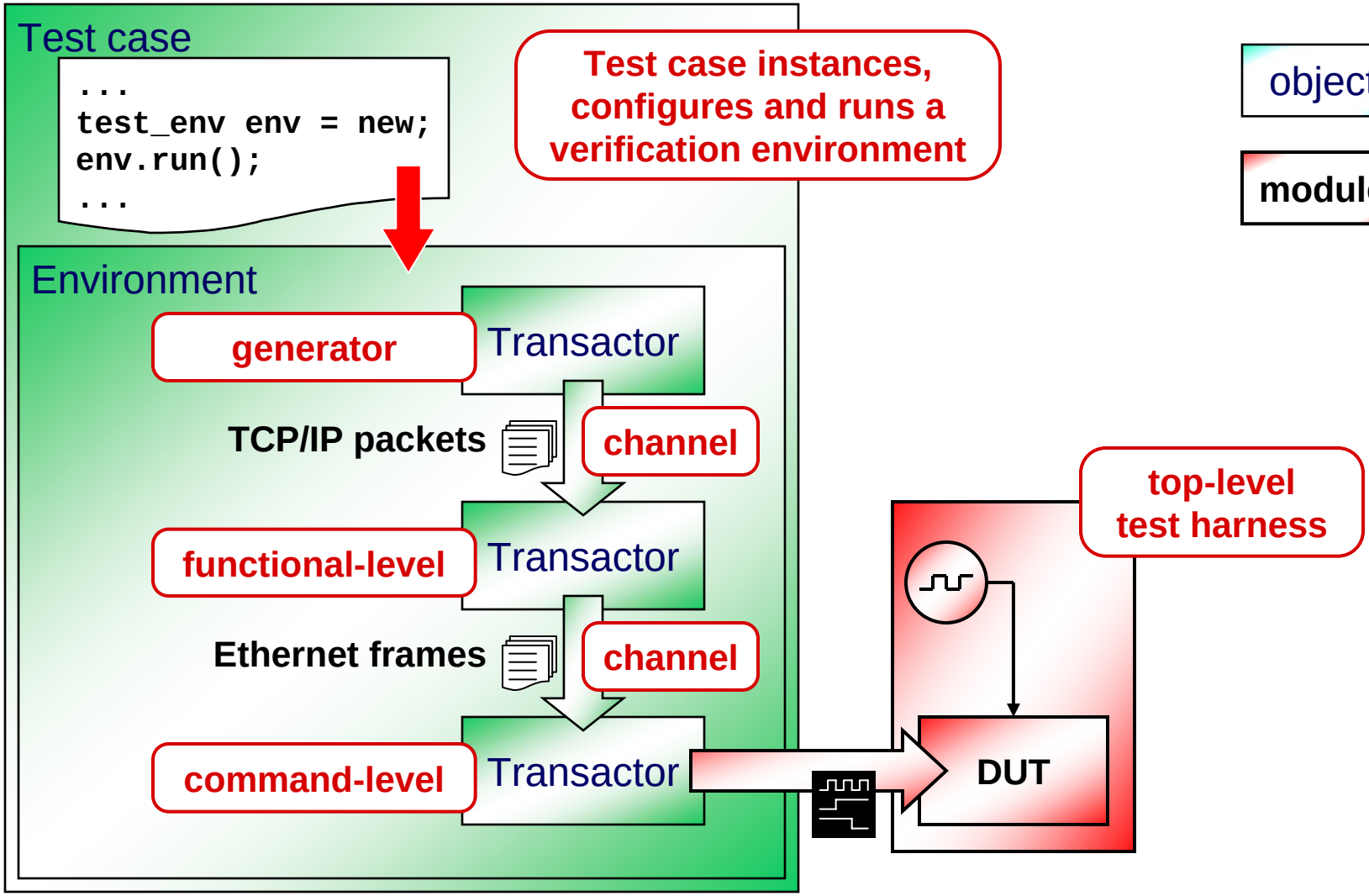
Transactors are class objects…

- … transactors are *not* module instances!

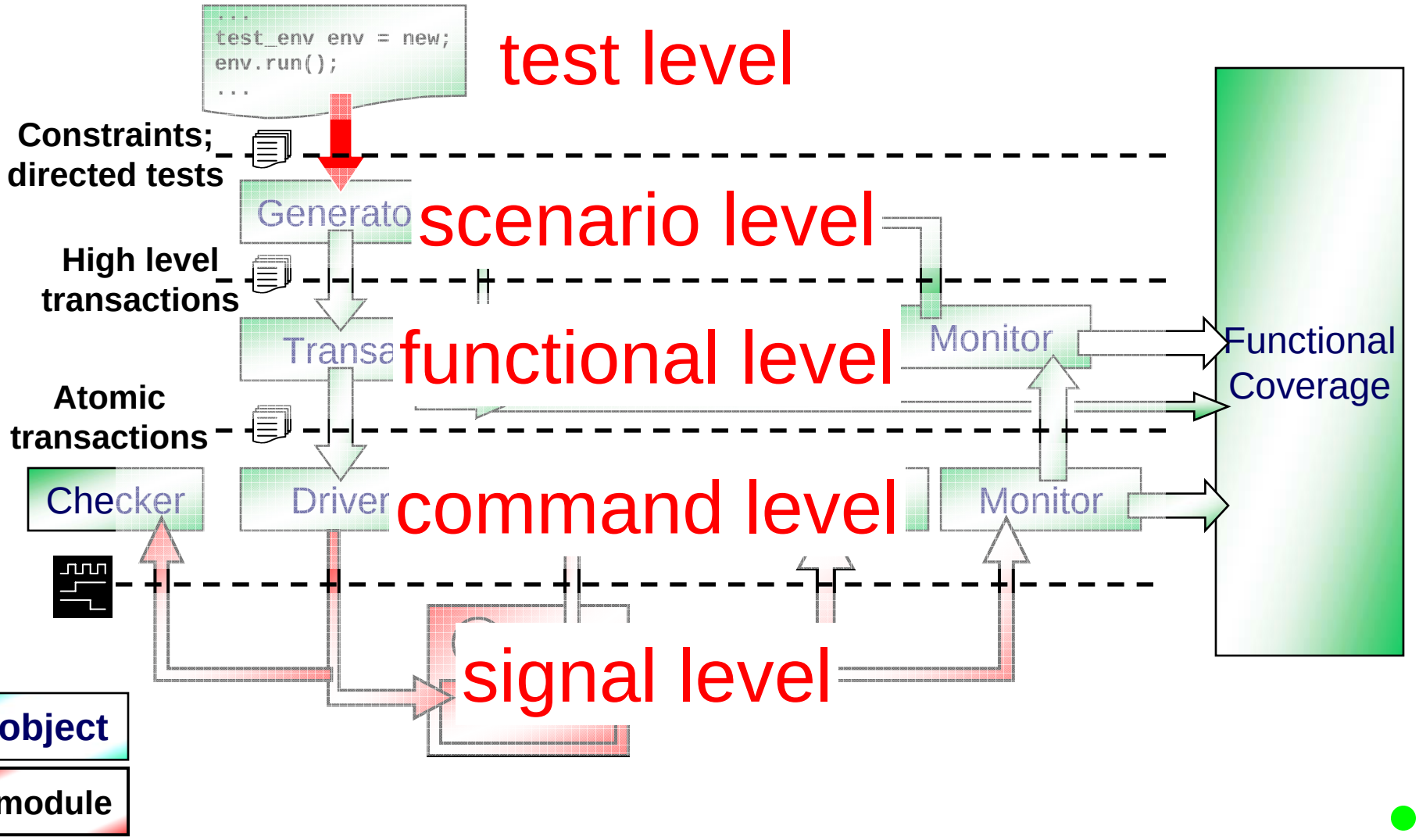
- Classes allow for
  - randomization of configuration of verification environment
  - OO inheritance
  - easy passing of object references

- Class objects can be instanced in a `program`
  - modules cannot

Test environment structure



Layered architecture



Agenda

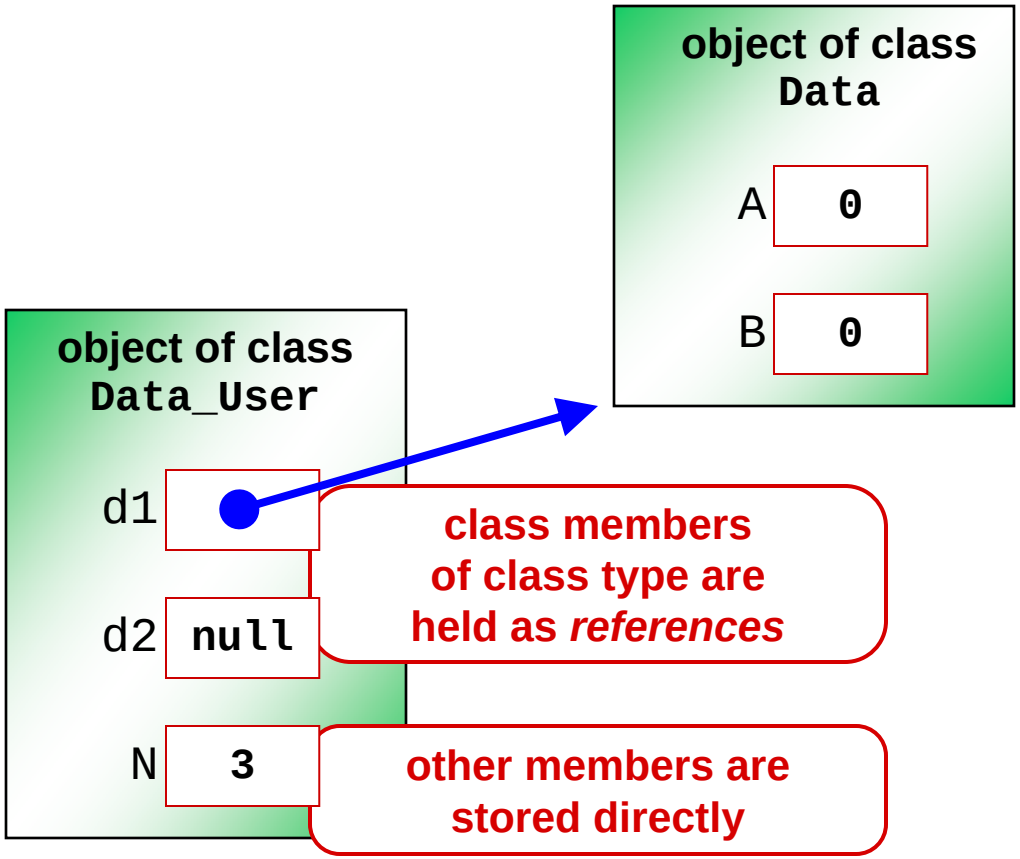
- Motivation and background
- Introduction to VMM test environment
- **SystemVerilog classes**
- Modeling test data
- Modeling the test environment
- Self-checking, assertions and coverage
- Creating testcases
- System-level verification (preview)
- Conclusions

Classes, objects and references

- It's not like C++…

```
class Data;
  rand int A;
  rand bit B;
endclass
```

```
class Data_User;
  Data d1 = new;
  Data d2;
  int N = 3;
endclass
```

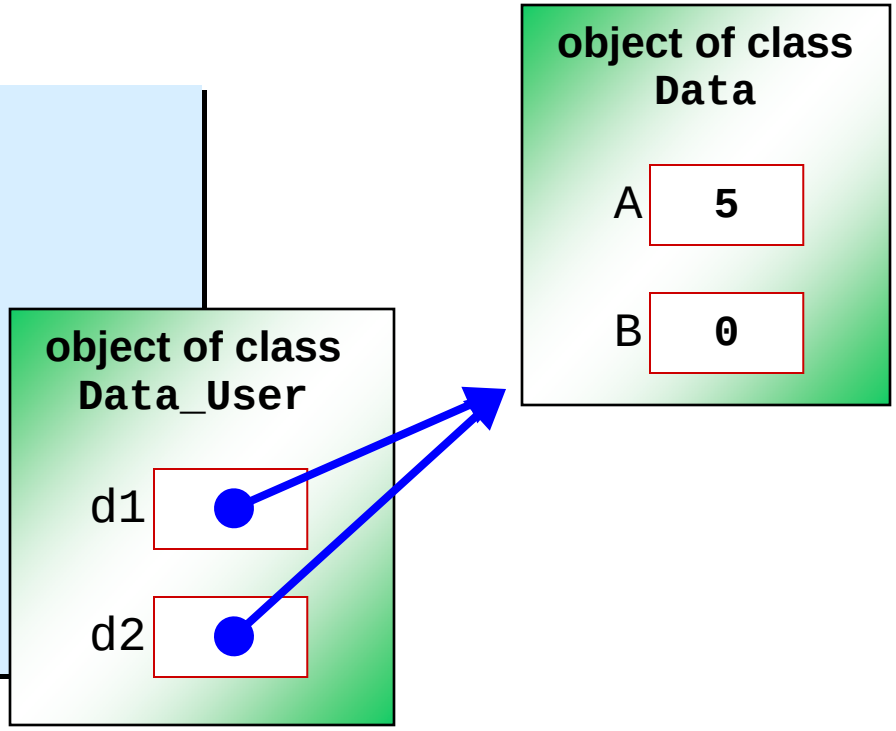


Beware copying!

- Copying a class-type member means copying the **reference**

```
class Data_User;
  Data d1 = new;
  Data d2 = d1;
  task confusing_copy;
    d1.A = 5;
    $display("A=%0d", d2.A);
  endtask
endclass
```

5

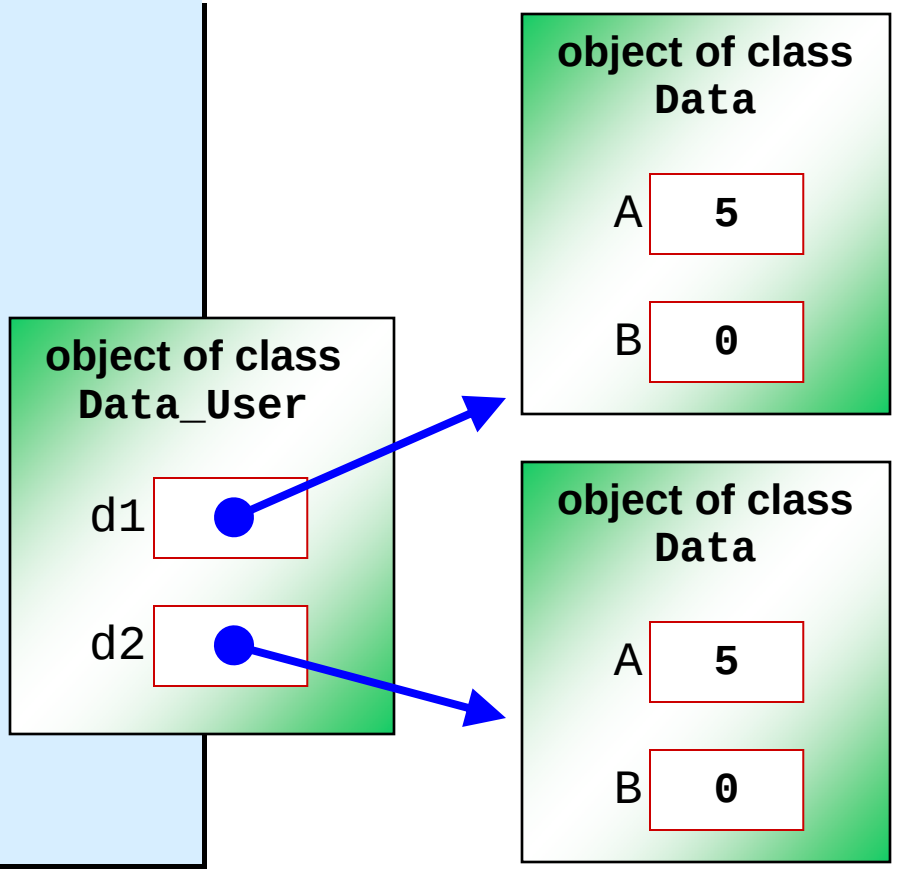


Beware comparison!

- Comparison means comparing the **references**

```
class Data_User;
  Data d1 = new;
  Data d2 = new;
  task confusing_compare;
    d1.A = 5;
    d2.A = 5;
    if (d1 == d2)
      $display("same");
    else
      $display("different");
  endtask
endclass
```

different



Solving the copy/compare problem

- Every useful class must have copy and compare methods
  - application-specific code to perform copy and comparison
  - virtual methods enforce calling convention
- VMM base classes provide this

Agenda

- Motivation and background
- Introduction to VMM test environment
- SystemVerilog classes
- **Modeling test data**
- Modeling the test environment
- Self-checking, assertions and coverage
- Creating testcases
- System-level verification (preview)
- Conclusions

Transaction data items

- Always modeled as a class derived from **`vmm_data`**
- All infrastructure expects to handle objects of the base class **`vmm_data`**
  - and can therefore handle derived-class objects too
- Core functionality is provided in the base class
  - Add your own data-specific extensions

Modeling transaction data

- Application-specific class derived from `vmm_data`

```
class Bus_Cycle extends vmm_data;
  typedef enum bit {MEM, IO} Bus_Space;
  typedef enum bit {RD,  WR} Bus_Dir;
  rand bit [15:0] addr;
  rand bit [ 7:0] data;
  rand Bus_Space  space;
  rand Bus_Dir    dir;
  ...
endclass : Bus_Cycle
```

data members representing transaction characteristics should be declared `rand`

standard virtual methods overridden here

Overriding virtual methods

extensions of `vmm_data` must override these

```
psdisplay
is_valid
allocate
copy
compare
```

optional

```
byte_pack
byte_unpack
byte_size
max_byte_size
```

```
rand bit [15:0] addr;
rand bit [ 7:0] data;
rand Bus_Space  space;
rand Bus_Dir    dir;
```

```
virtual function bit is_valid (
    bit silent = 1,
    int kind = -1 );
  if ((space == IO) && (addr > 0xff))
    is_valid = 0;
  else
    is_valid = 1;
  if (!is_valid && !silent)
    <generate error message>
endfunction
```

vmm_data::copy()

```
virtual function vmm_data copy (vmm_data to = null);
```

- Copies this object into the target (`to`) object
  - returns reference to copied object
- If target is null, automatically creates new object
  - using virtual method `allocate()`, not new!
- Checks target is of appropriate derived class

- Recommended implementation shown in VMM Standard Library Specification

vmm_data::copy_data()


```
virtual protected function void copy_data (vmm_data to);
```

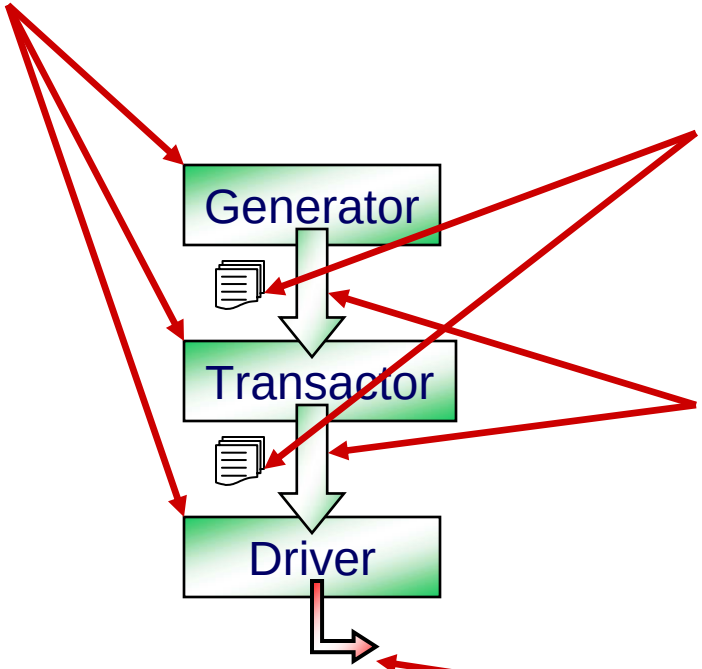
- Copies the vmm_data base-class contents
  - You don't need to know the details of what's in vmm_data

Agenda

- Motivation and background
- Introduction to VMM test environment
- SystemVerilog classes
- Modeling test data
- **Modeling the test environment**
- Self-checking, assertions and coverage
- Creating testcases
- System-level verification (preview)
- Conclusions

Modeling transactors and channels

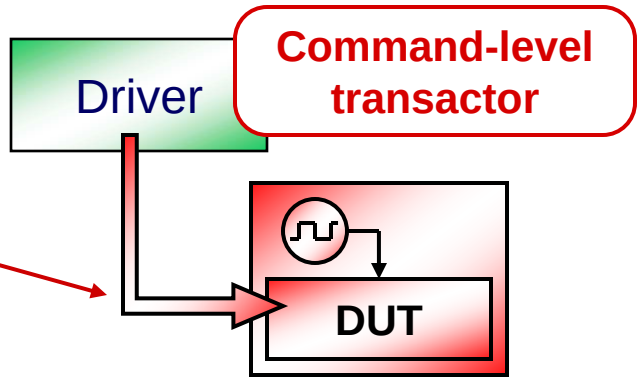
- All transactors derived from `vmm_xactor`



- Transaction objects modeled using `vmm_data`
- Channels created using `` `vmm_channel `` macro
- Pin-level interconnect modeled as `interface`

Pin level interconnect

- Driver is a class object, *not a module instance*
  - Allows randomized configuration of test environment
- Pin-level interconnect modeled as `interface`



- How can we connect a driver to *any* suitable interface instance?
  - Solution: *virtual interface*

Virtual interface

```
class cmd_level_xactor extends vmm_xactor;
  virtual interface itf.clx V;
  task pulse;
    V.C.sig <= 1'b1;
    @(V.C);
    if (V.C.sig) ...
  endtask
endclass
```

Transactor can drive any instance of this interface

Cycle delay

```
interface itf (input bit clk);
  wire sig;
  clocking C @(posedge clk);
    output #1 sig;
  endclocking
  modport clx (clocking C);
endinterface
```

Clocking block specifies directions and detailed timing

Modport gives access to clocking block

Connecting the virtual interface

```
class cmd_level_xactor extends vmm_xactor;
  virtual interface itf.clx V;
  function new (virtual interface itf.clx V);
    this.V = V;
    ...
  endfunction
  ...
```

constructor

```
module top;
  bit clock;
  itf Drv_Itf (clock);
  ...
```

test harness

```
class test_env extends vmm_env;
 ...
  virtual function void build();
    ...
    cmd_level_xactor Driver =
            new(top.Drv_Itf.clx);
```

interface instance supplied as constructor argument

Execution environment

- Transactor code must execute in Reactive region
  - Avoids races with design, uses `clocking` correctly
- Instantiate class and call its code from a `program`

```
module bad_top_level;
  initial begin
    cmd_level_xactor
        BFM(top.Drv_Itf.clx);
    ...
    BFM.pulse();
    ...
```

X

X

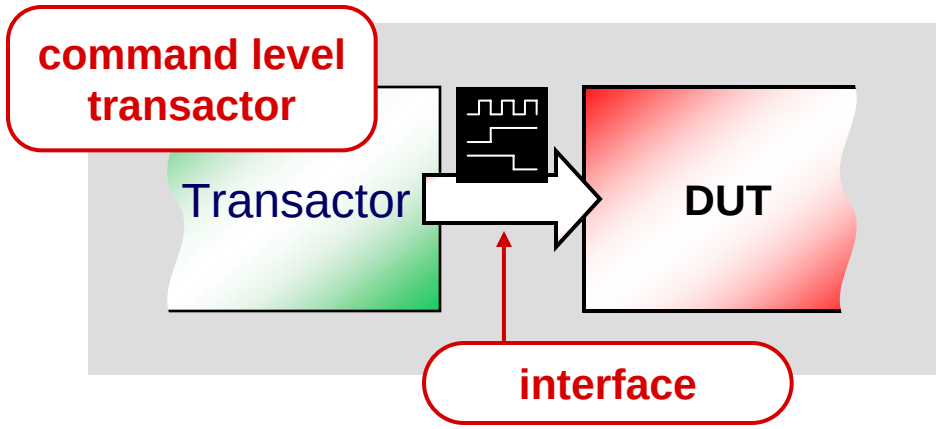
```
program good_top_level;
  initial begin
    cmd_level_xactor
        BFM (top.Drv_Itf.clx);
    ...
    BFM.pulse();
    ...
```

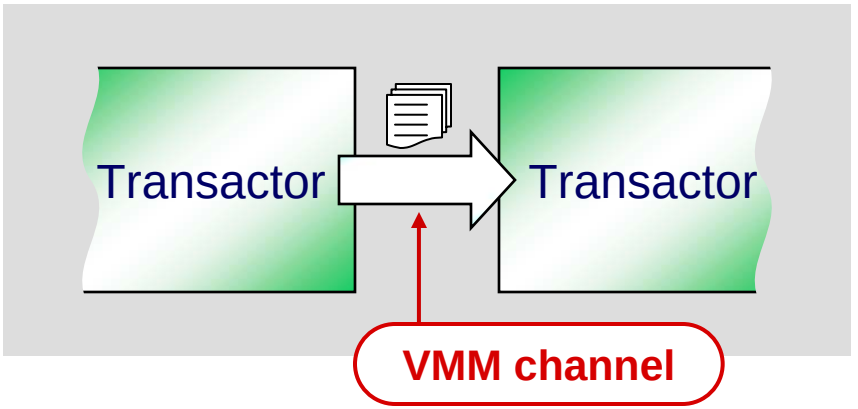
OK

OK

Transactors and channels (*review*)

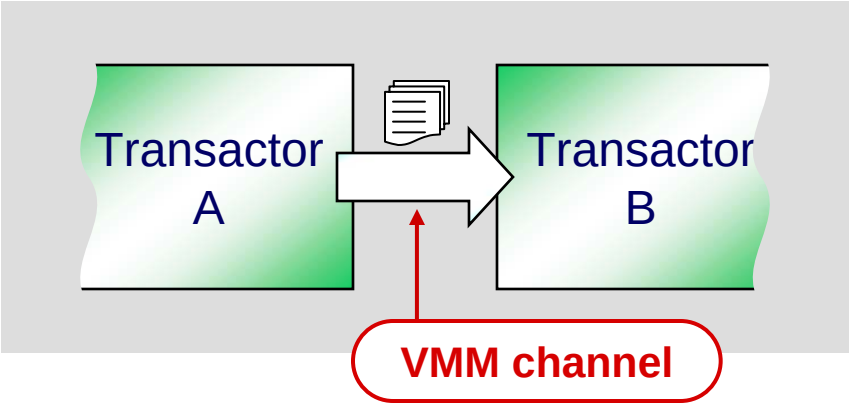
- Some transactors connect to signals …
- … some connect to other transactors





Why use channels?

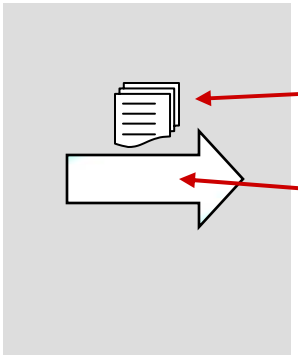
- Transactor A could call task/function in Transactor B
  - Each transactor would need to know about its neighbour



- Channel decouples each transactor from others
  - Uniform protocol
  - Transactors can be swapped easily

Creating a channel

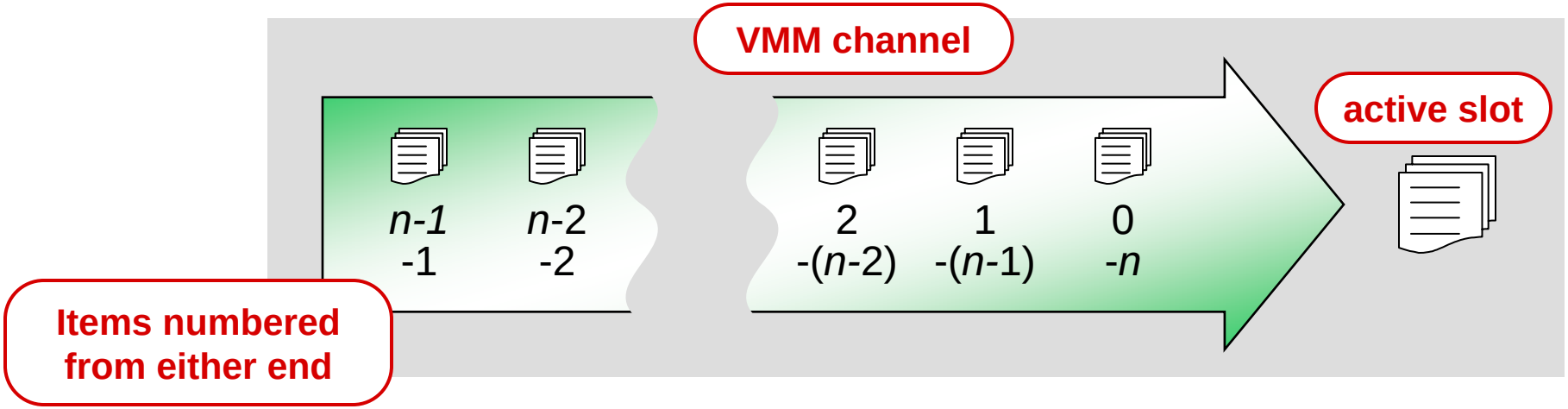
- Create a channel class for each transaction class
  - Independent of any connected transactors



```
class Bus_Cycle extends vmm_data;
  ...
endclass : Bus_Cycle
`vmm_channel(Bus_Cycle);
```

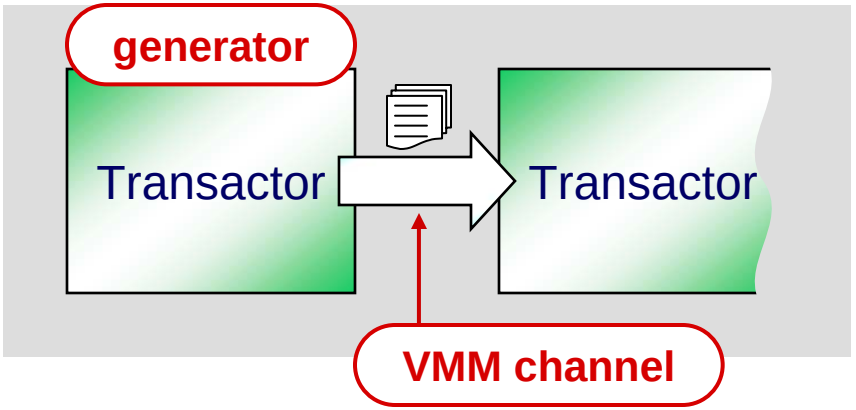
- Macro `vmm_channel hides implementation detail
  - Creates new class Bus_Cycle_**channel**

Channel features



- `put/get/peek` functions may block on empty/full
- `sneak` function inserts an entry even if channel full
- `peek` function examines without removing
- *active slot* stores “item currently being consumed”

Generators



- A transactor with no upstream input
- Typically sends data to a channel
  - best to keep generator and command-level distinct
- Needs a generation strategy
  - Single transactions? Streams of transactions?

Creating an atomic generator

```
class Bus_Cycle extends vmm_data;
  ...
endclass : Bus_Cycle
`vmm_channel(Bus_Cycle);
`vmm_atomic_gen(Bus_Cycle, "description");
```

defines class `Bus_Cycle_channel`

defines class `Bus_Cycle_atomic_gen`

- Data class *must* have constructor with no arguments
- Generator streams data items into its output channel
  - Data items are unrelated

Additional features of atomic generator

- Generator class is inherited from `vmm_xactor`
  - start, stop, reset functionality
  - callbacks (more detail later!)

- `stop_after_n_insts`

- `inject` method allows testcase to insert directed test items amongst the random stream

Randomisation

- Data members of `vmm_data` derivatives should be declared `rand`
  - allows random generation via `randomize()` method
  - derived classes can add constraints
  - constraints can be switched using `constraint_mode()`
  - item randomization can be switched using `rand_mode()`

- How do you randomize *which* derived class to use?

Factory

- Why go to all this trouble?

```
class Bus_xactor extends vmm_xactor;
  Bus_Cycle randomized_cycle;
  ...
  function bit make_a_cycle();
    randomized_cycle.randomize();
    return randomized_cycle.copy();
  endfunction : make_a_cycle
  ...
endclass : Bus_xactor
```

Using a factory

- It's easy to upgrade:

```
class Bi_Cycle extends Bus_Cycle;
  ...
endclass
```

```
class Bus_xactor extends vmm_xactor;
  Bus_Cycle randomized_cycle;
  ...
  function bit make_a_cycle();
    randomized_cycle.randomize();
    return randomized_cycle.copy();
  endfunction : make_a_cycle
  ...
endclass : Bus_xactor
```

```
task Special_Test;
  Bus_xactor gen;
  Bus_Cycle Bus_factory = new;
  Bi_Cycle Bi_factory = new;
  ...
  gen.randomized_cycle = Bi_factory;
  ... // do some tests
  gen.randomized_cycle = Bus_factory;
  gen.randomized_cycle.Constr1.constraint_mode(0);
  ... // do some more tests
endtask : Bi_cycle
```

control individual constraints

Creating a scenario generator

```
class Bus_Cycle extends vmm_data;  ...
```

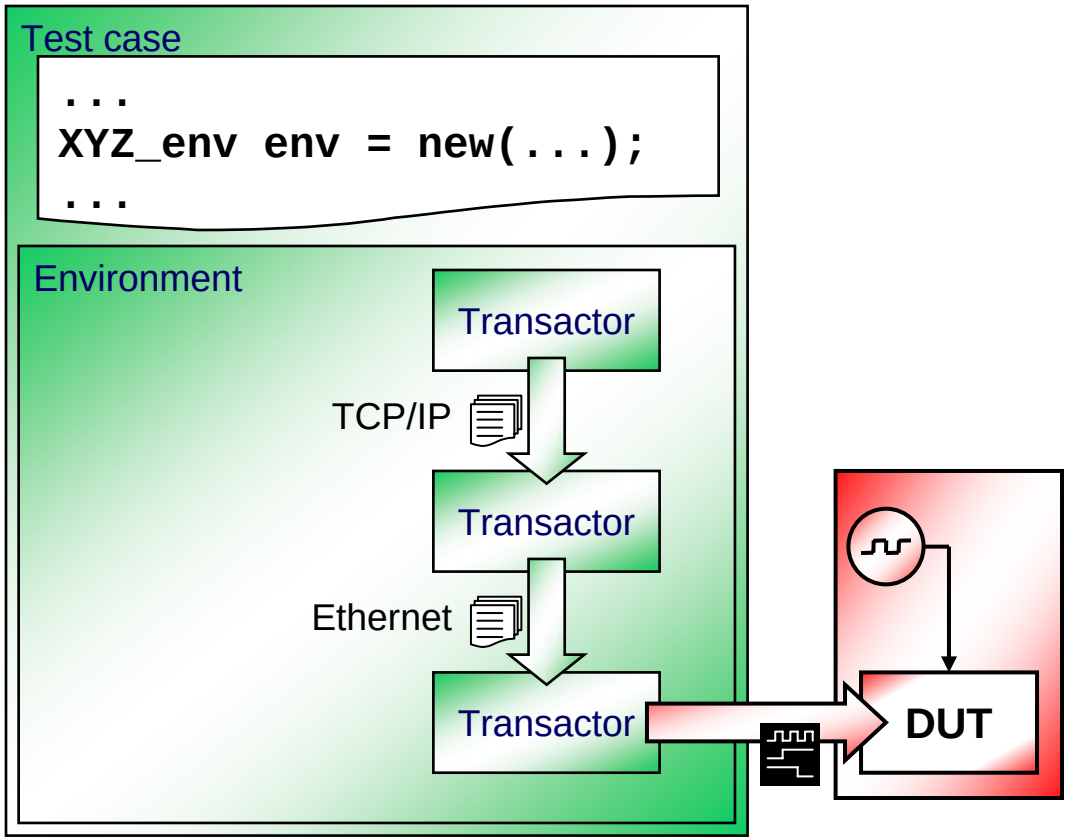
```
`vmm_scenario_gen(Bus_Cycle, "description");
```

- Defines several new classes:
  - `Bus_Cycle_scenario` — randomizable scenario descriptor
  - `Bus_Cycle_atomic_scenario` — simple scenario with just one cycle
  - `Bus_Cycle_scenario_gen` — derivative of `vmm_xactor`
  - `Bus_Cycle_scenario_election` — makes random choice of scenario
  - `Bus_Cycle_scenario_gen_callbacks` — callback façade

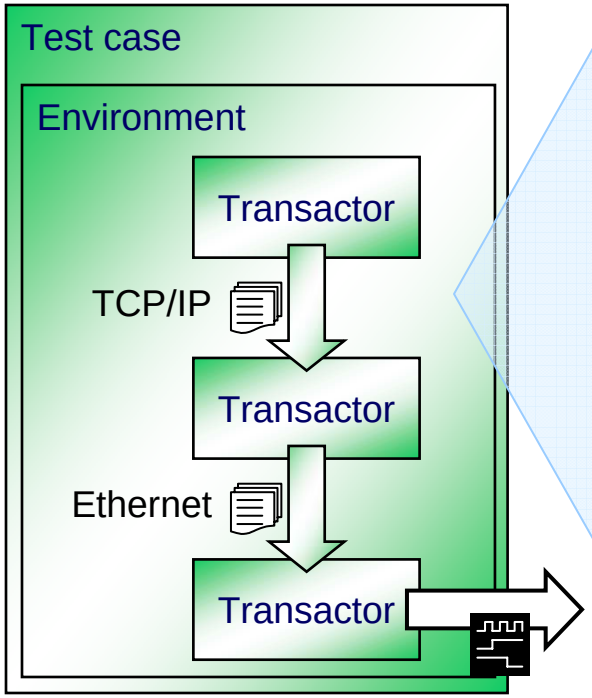
Building the environment

- Testcase instantiates the specialised verification environment

- Environment has virtual methods that configure, build and start everything



Environment virtual methods



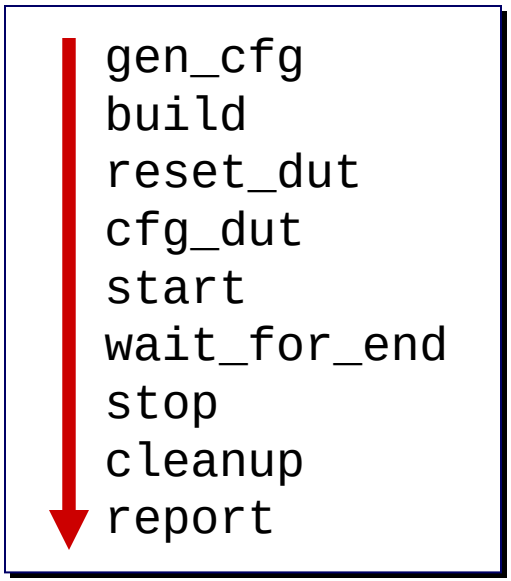

```
class XYZ_env extends vmm_env;
  TCPIP_xactor PktGen;
  ETH_xactor   FrameGen;
  MII_xactor   MII_drv;
  ...
  virtual function void build();
    super.build();
    PktGen = new(...);
    FrameGen = new(...);
    ...
  endfunction : build
  ...
endclass : XYZ_env
```

override standard methods

call base-class methods

customize environment

Virtual methods of `vmm_env`



- Extensions of `vmm_env` must override these methods
- Extensions must first call base-class method using **`super.`**
  - guarantees correct operation of `run()`

- `gen_cfg()` randomizes a configuration descriptor
- `build()` constructs environment as specified by configuration descriptor

Concessions to support legacy code

- Command-level transactors can be re-packaged as `module`
  - use VMM-compliant blocks as BFM in traditional testbench
  - transactor and atomic generator instantiated and started automatically
  - controlled through procedural interface

- Existing BFMs can be re-packaged as transactors
  - eases migration from legacy environment to VMM compliance

Detailed recommendations at the end of VMM chapter 4

Agenda

- Motivation and background
- Introduction to VMM test environment
- SystemVerilog classes
- Modeling data
- Modeling the test environment
- **Self-checking, assertions and coverage**
- Creating testcases
- System-level verification (preview)
- Conclusions

Assertions

- VMM offers extensive guidance
- Useful for…
  - protocol checkers
  - sequencing functionality checks (*e.g.* arbiter - see SVA checker library)
  - protocol coverage using `cover property`
  - triggering a covergroup using `sample()`

Checkers

- Build assertion-based checkers into an `interface`
  - Allows instantiation into `program`, `interface` or `module`
- Libraries of pre-existing checkers
  - OVL-equivalent with enhanced instrumentation
  - Advanced checker library (currently Synopsys proprietary)
- Or write your own using SVA

Scientific verification

- What did I measure?

  COVERAGE

- What did I learn from the measurements?

  Assertions and error checkers

- What could I do better next time?

  Change tests to improve coverage

- Have I measured enough?

  Coverage relative to goals

Coverage

- Properties written to check protocols can also act as coverage points

```
property cycle_start;
  !ALE ##1 ALE;
endproperty
property cycle_OK;
  cycle_start |-> (RD ^ WR) [*1:3] ##1 !RD && !WR;
endproperty

assert property (cycle_OK);

cover property (cycle_start);
```

Identify the start of a read or write cycle

Identify correctness of a read or write cycle

Check for correct protocol

Count cycles

Controlling coverage and assertions

- Standard macros for global control via `ifdef
  - `ASSERT_ON
  - `COVER_ON
  - `NO_INLINE_ASSERTIONS

- Detailed control via parameter coverage_level_*n* in each class that implements functional coverage
  - one bit for each coverage point in the class
  - if that bit is set in coverage_level_*n*, a conditional generate inserts appropriate coverage code

***n* = 1, 2, 3**

Agenda

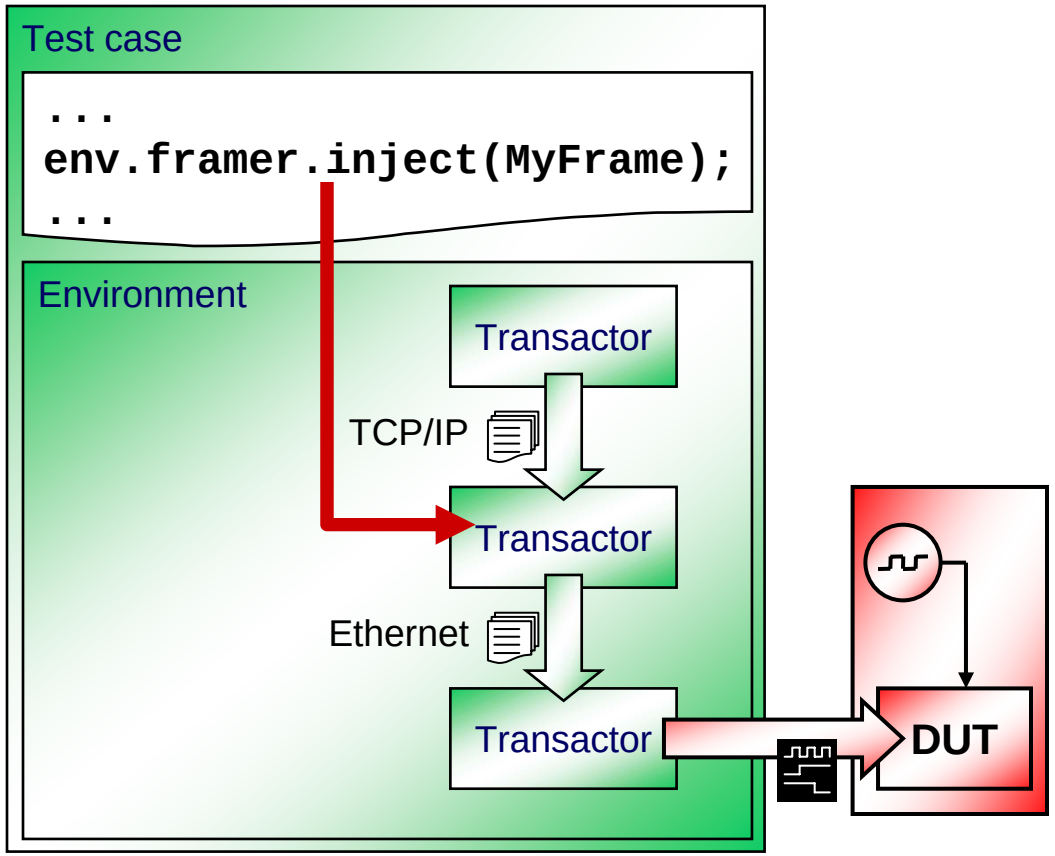
- Motivation and background
- Introduction to VMM test environment
- SystemVerilog classes
- Modeling test data
- Modeling the test environment
- Self-checking, assertions and coverage
- **Creating testcases**
- System-level verification (preview)
- Conclusions

Testcase authoring

- Testcase is top level
  - Instantiates the verification environment
- Testcase can configure environment and modify its constraints
- Testcase can inject specific directed stimulus
  - e.g. for register initialisation in DUT, corner case testing
- Testcase can hook into, and modify, any transaction
  - callback mechanism
  - e.g. for error injection, monitoring

Injecting directed tests

- Every transactor provides an `inject()` method
  - User-written test case can insert arbitrary stimulus at any level of abstraction
- Injected transactions trigger callbacks
  - as if they were from the transactor



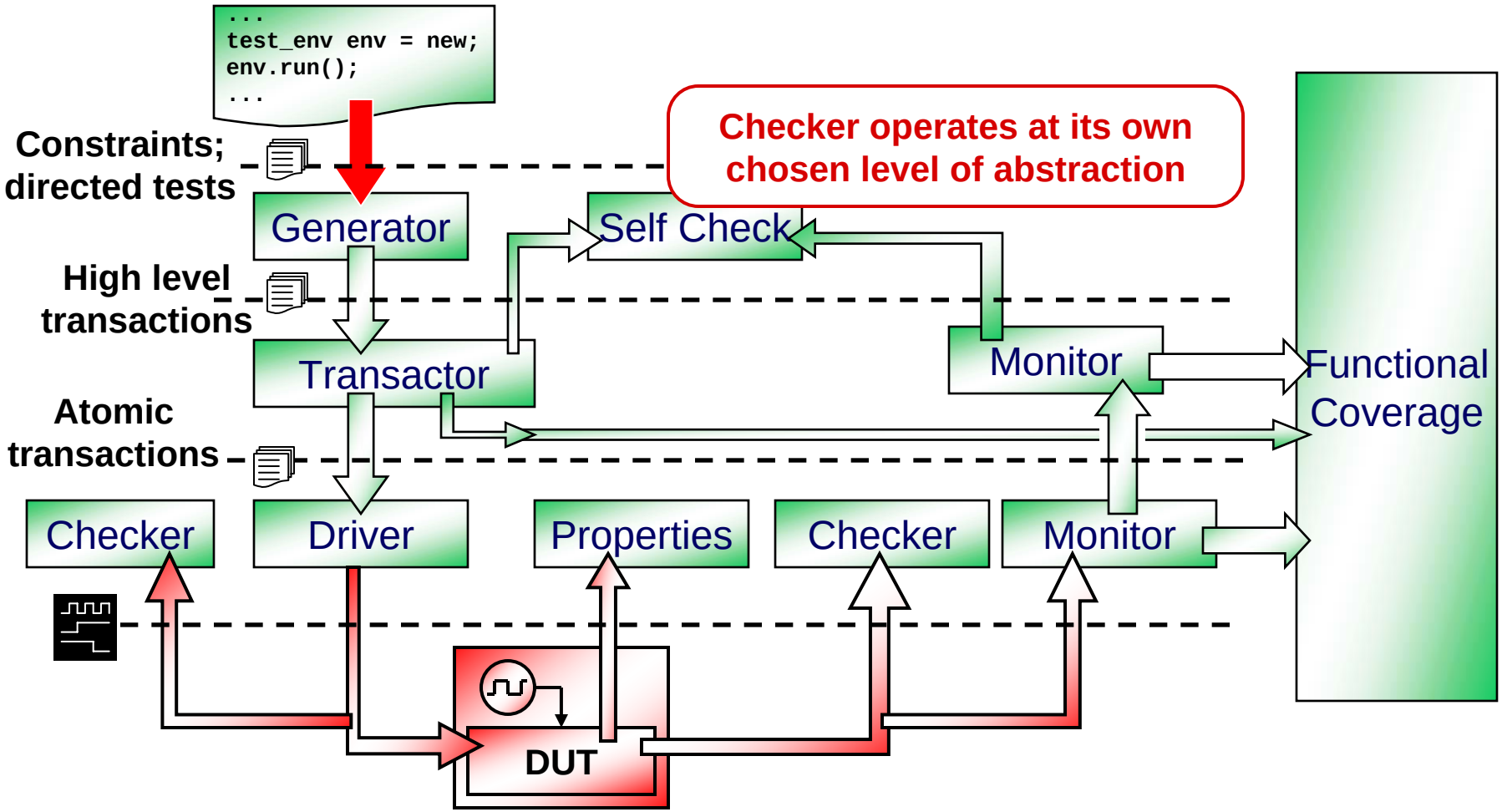
Checking overall behavior

- Not part of the testcase's problem, but…
- … another reason to use callbacks!

- Every transaction should be remembered so its results can be checked **Scoreboard**

- Scoreboard and checking should be independent of transactor

Checker in the architecture



Callbacks

- Transactor should call an empty function whenever it does anything interesting
- Extensions supply a body for the empty function
  - to hook into the transactor's activity
- A great idea, but hard to make it work…
  - Where is the empty function defined? How is it extended?
  - Dynamically add/remove these hooks?

Solution: The callback façade class

- Transactor

```
class test_xactor extends vmm_xactor;
  ...
  do interesting stuff;
  `vmm_callback(
       test_callbacks,
       interesting(this, ..., ...) );
  ...
endclass : test_xactor
```

must use this macro

façade class name

façade function to call

Reference to calling transactor

- Façade

```
class test_callbacks extends vmm_callbacks;
  virtual function void interesting (
       test_xactor source, ..., ...);
  endfunction
endclass : test_callbacks
```

Must be task or function void

Specialising a callback façade

- User-defined extension functionality
  - triggered by callback in transactor

```
class test_checker_callbacks extends test_callbacks;
  virtual function void interesting (
    test_xactor source, ..., ...);
      // make use of or modify the information
  endfunction
endclass : test_checker_callbacks
```

- Pass information back to transactor
  - through `ref` or non-`const` object arguments

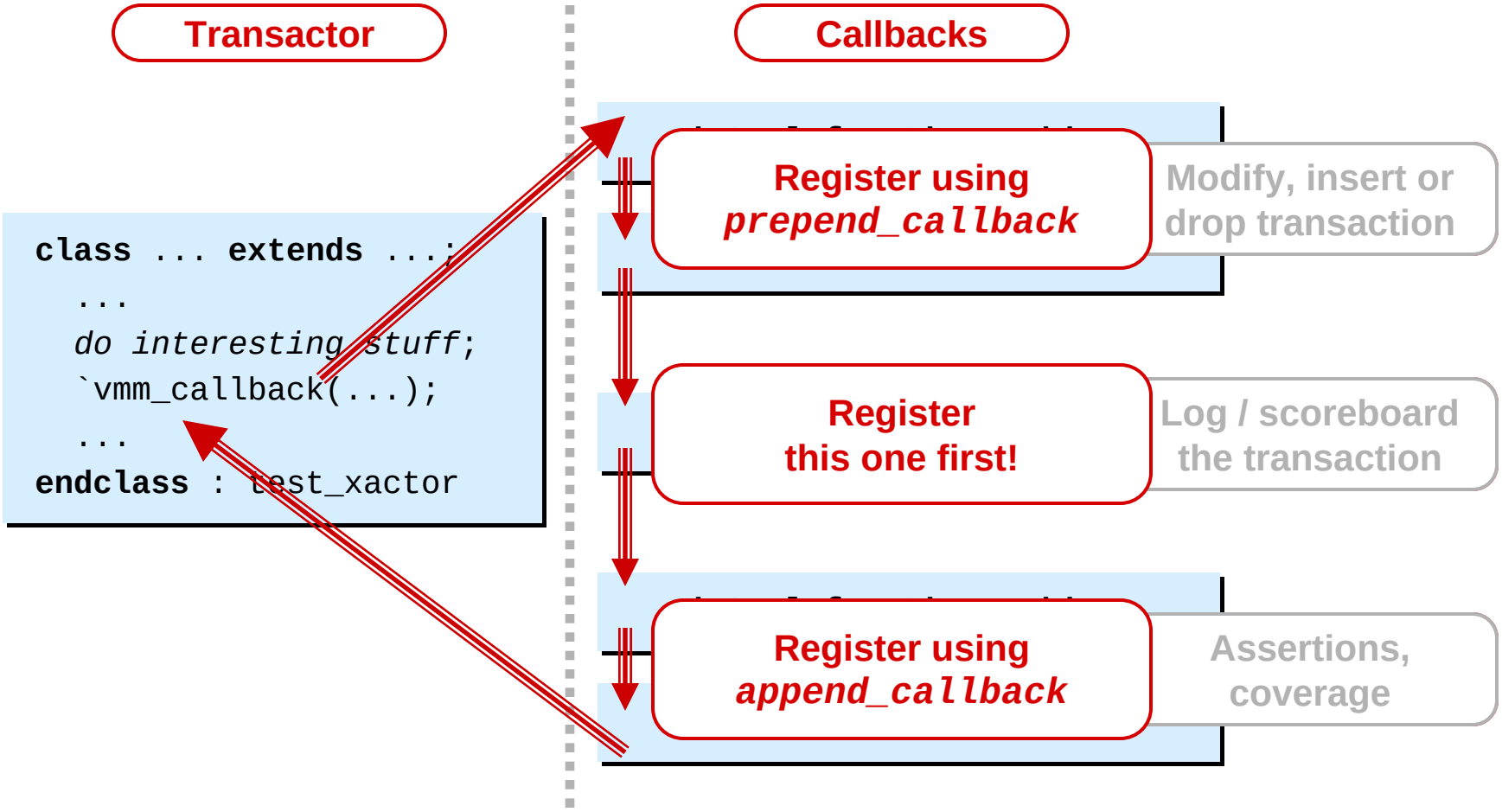
Using a specialised callback façade

- Environment's `build()` method registers callbacks

```
class my_special_env extends vmm_env;
  test_xactor stim_src;
  test_checker_callbacks cb;
  ...
  virtual function void build();
    ...
    cb = new(...);
    stim_src.append_callback(cb);
    ...
  endfunction : build
  ...
endclass : my_special_env
```

Builds environment according to *configuration descriptor*

Order of execution of callbacks



Agenda

- Motivation and background
- Introduction to VMM test environment
- SystemVerilog classes
- Modeling test data
- Modeling the test environment
- Self-checking, assertions and coverage
- Creating testcases
- **System-level verification (preview)**
- Conclusions

XVCs and coordination of stimulus

Just an introduction!

- ARM's system-level verification architecture
- XVC = Extensible Verification Component
- Like a transactor, but transactions controlled from an *action queue*
  - Actions can be derived from a plain-text command file
- Multiple XVCs can be coordinated by a single XVC Manager instance
  - facilitates generation of scenarios with specific timing relationships between different activities

Agenda

- Motivation and background
- Introduction to VMM test environment
- SystemVerilog classes
- Modeling test data
- Modeling the test environment
- Self-checking and assertions
- Generating stimulus
- System-level verification (preview)
- **Conclusions**

Current levels of support

- Appendix A of the book provides a complete description of VMM Standard Library, enabling engineers to create their own implementation
- Synopsys provides a compiled version of VMM Standard Library with VCS, to speed adoption
- Synopsys will license the source code of its implementation of VMM Standard Library to customers and SystemVerilog Catalyst members

Conclusions

- VMM packages best practice ready for you to use:
  - SystemVerilog issues already handled
  - scalable for very large environments
  - successfully assimilates legacy verification code
- VMM Standard Library benefits:
  - encourages interoperability
  - provides initial framework
  - out-of-the-box solutions for many problems
- Backed-up by a detailed and practical book

Coda

- Thanks!
- Questions?