Verification

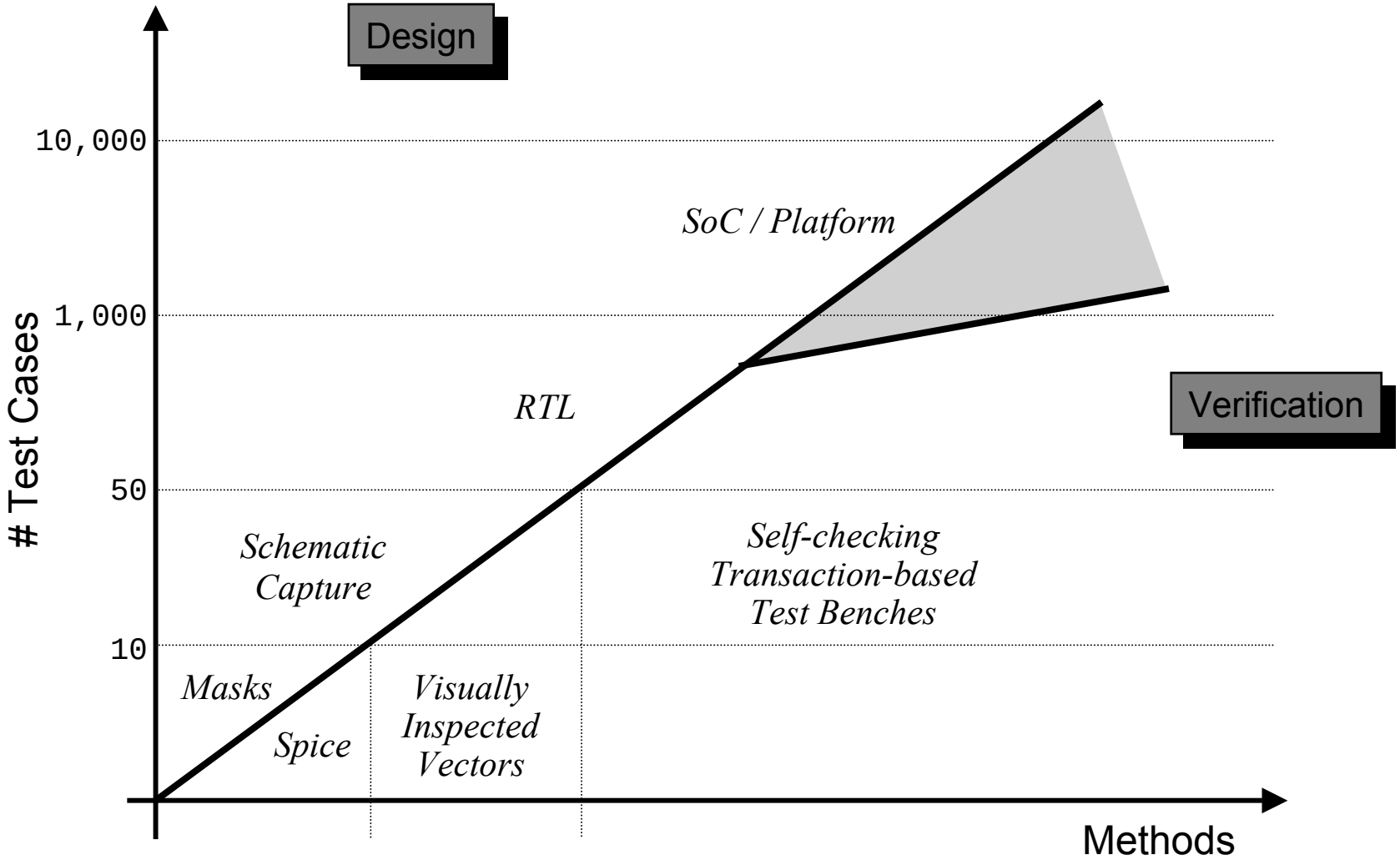
Verification Methodology using Vera or SystemVerilog

SNUG BOSTON 2003

Janick Bergeron
Tom Fitzpatrick

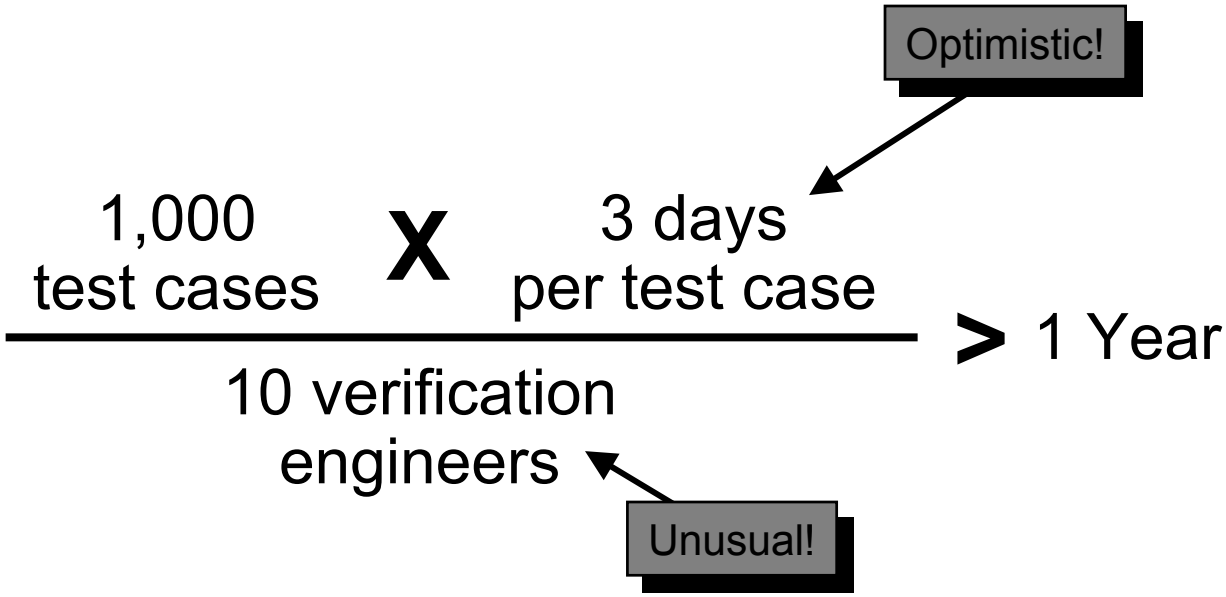
Verification Group
Synopsys, Inc.

Verification Productivity Gap



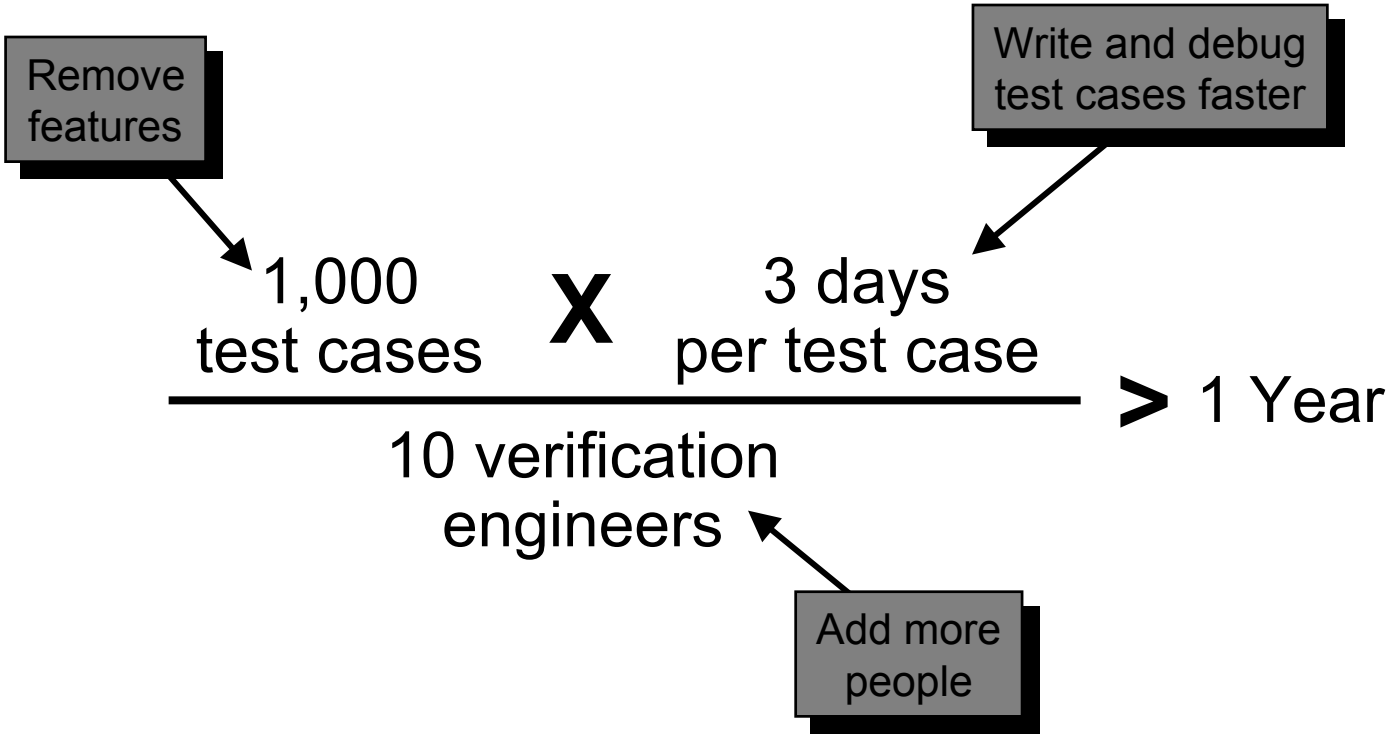
Verification Productivity Gap

Verification Effort



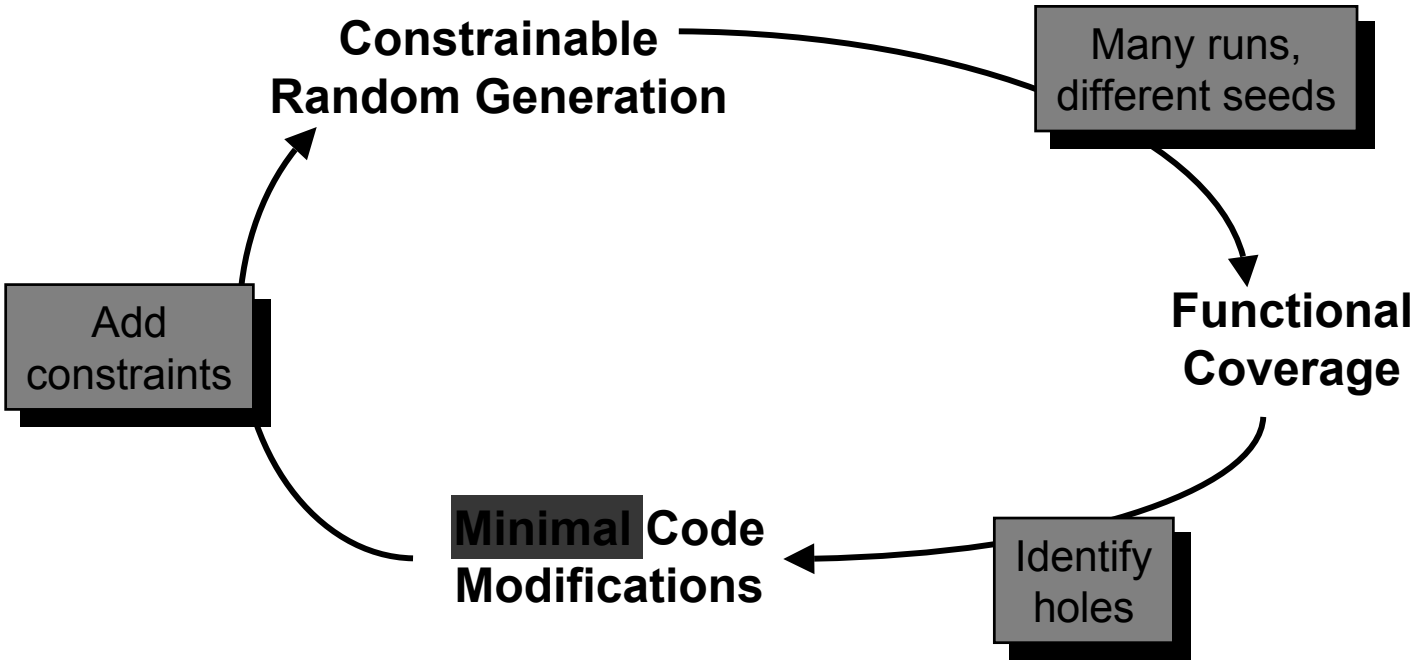
Verification Productivity Gap

Improving Productivity



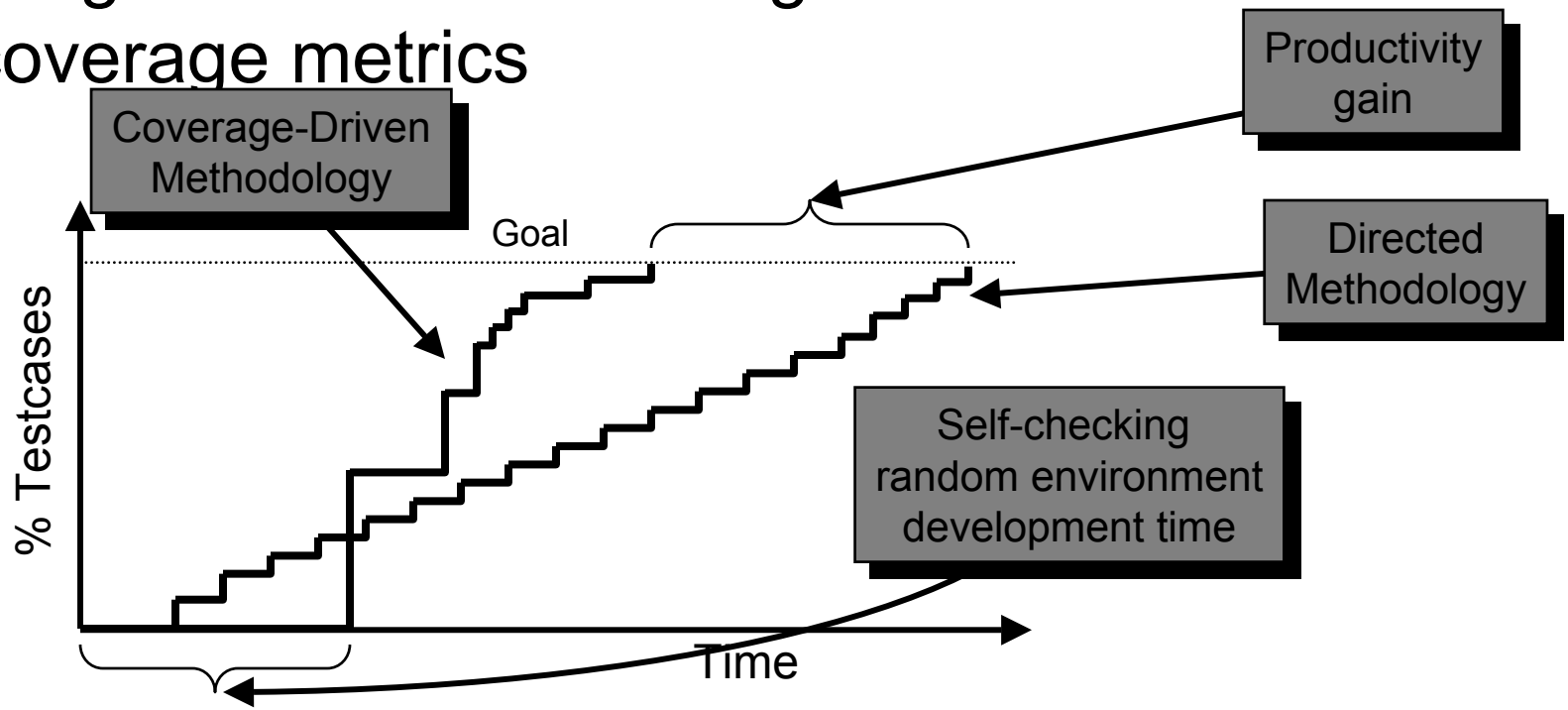
Coverage-Driven Verification

- Trade-off authoring time for run-time
- Focus on uncovered areas

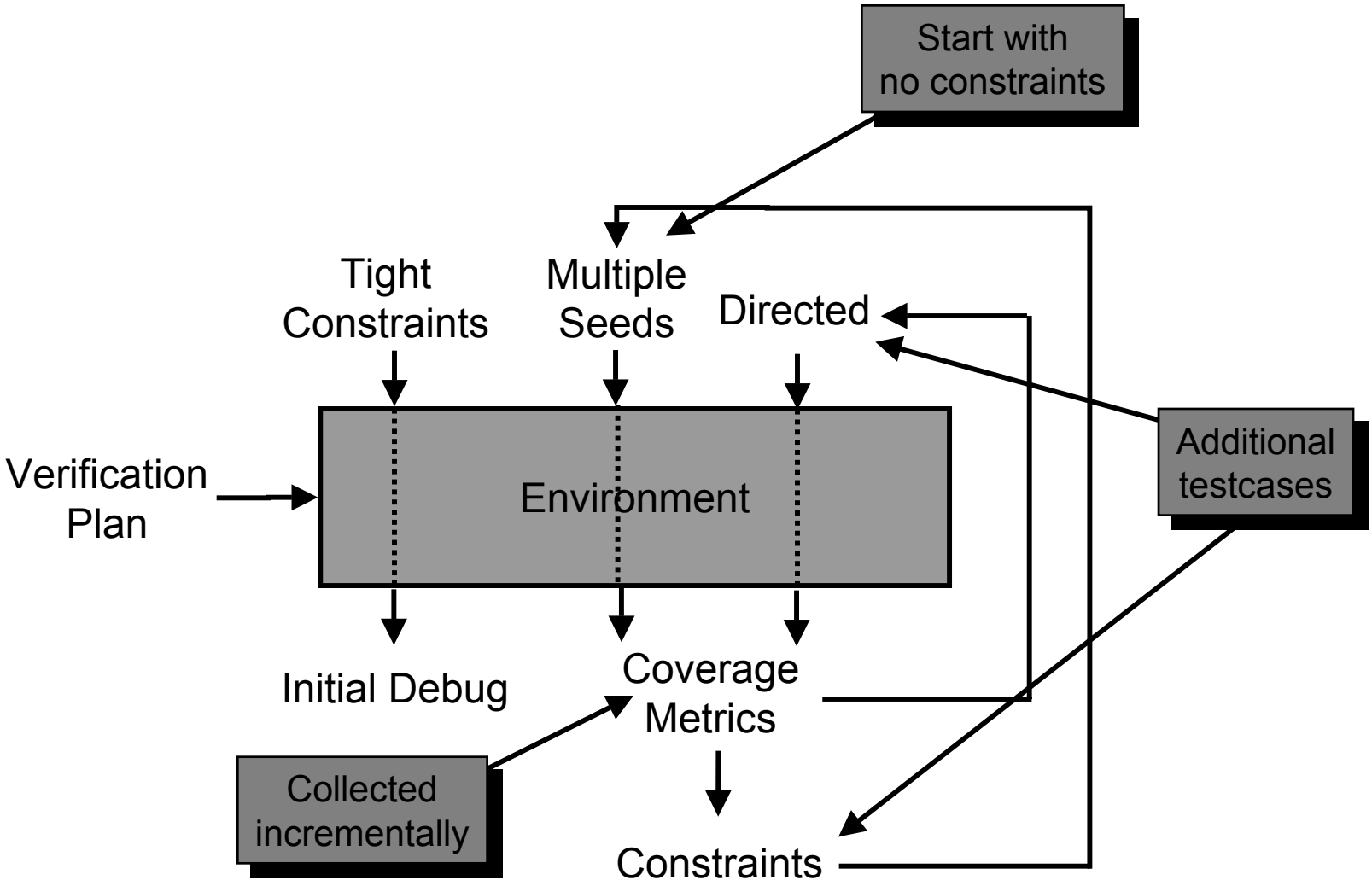


Coverage-Driven Verification

- Minimal number of test environments
  - Significant number of test files
  - Large number of test runs
- Progress measured using functional coverage metrics

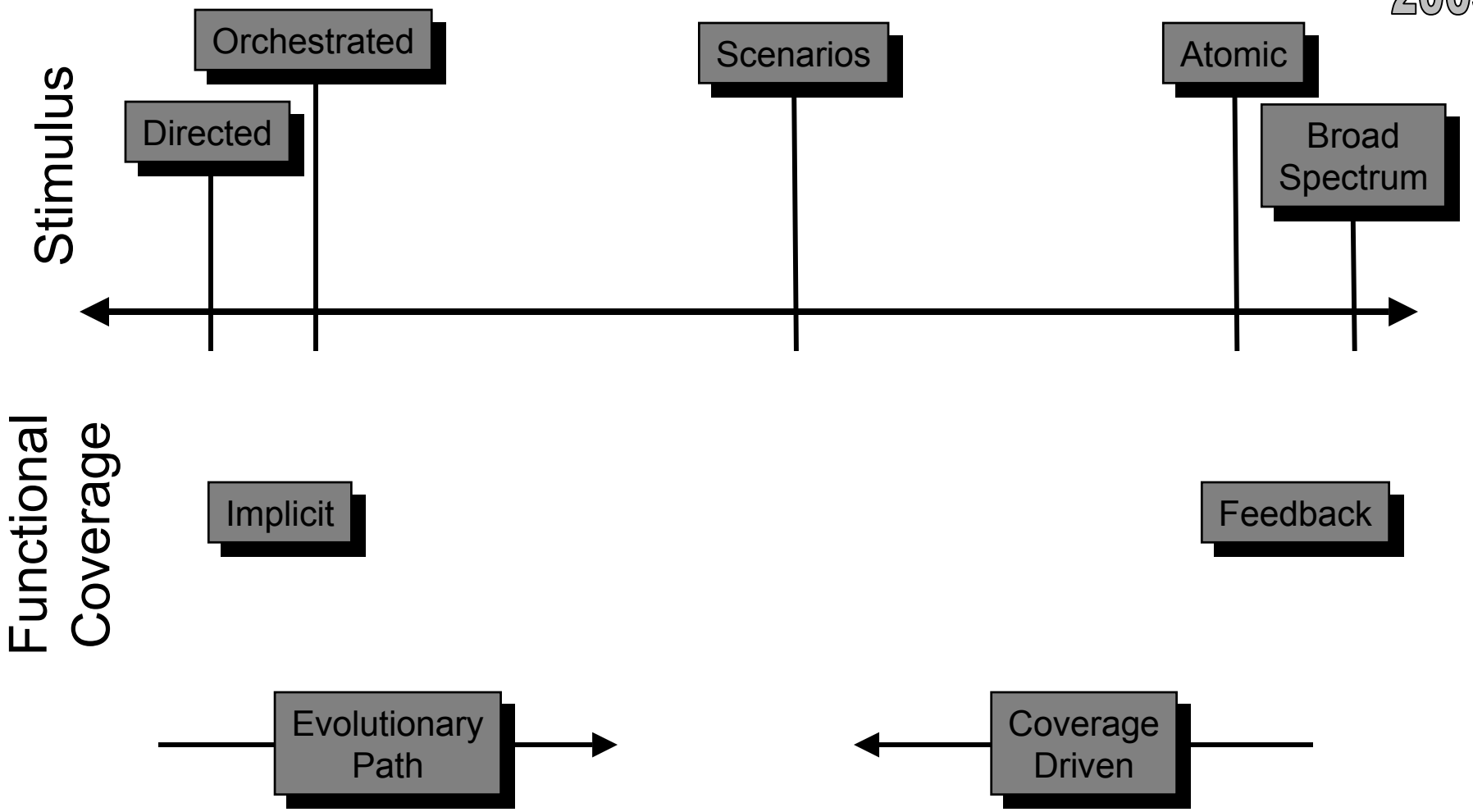


Coverage-Driven Verification



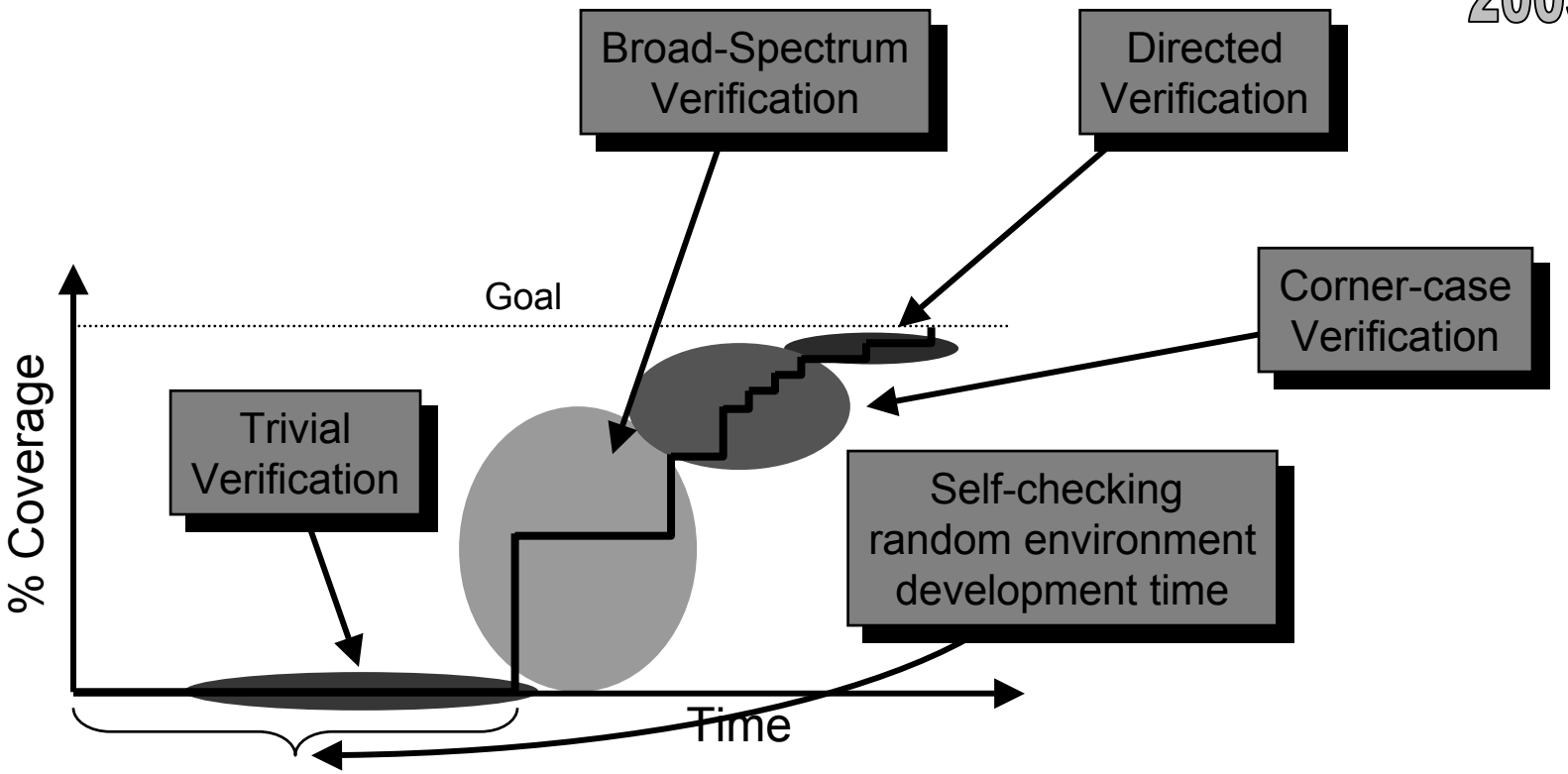
Stimulus Spectrum

Randomness



Stimuli Development

Timeline



RVF Methodology

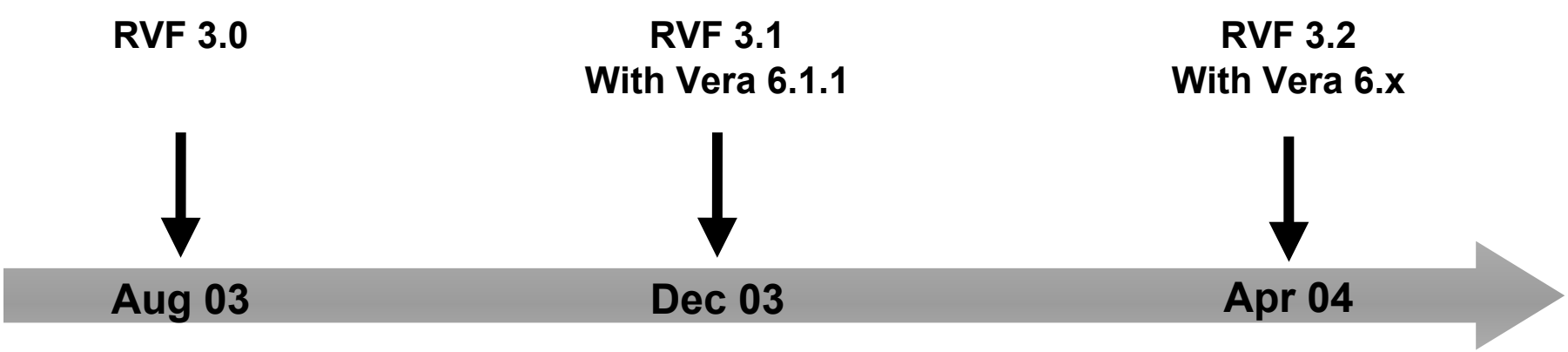
Objectives

- Support coverage-driven as well as directed verification
- Help customers realize full benefits of Synopsys verification tools faster
  - Languages
  - Constraints and solver
  - Coverage
  - Assertions
  - Formal proofs
- Promotes creation of reusable data models, transactors and generators

Methodology
Guiding Principles

- Maximize design quality
  - More testcases
  - More checks
  - Less code
- Approaches
  - Reuse
    - Across projects
    - Across blocks
    - Across systems
    - Across tests
  - One verification environment, many tests
  - Minimize test-specific code

RVF Methodology
Deliverables



- RVF Methodology Guidelines
- RVF Base Classes
- Training Materials
- Simple example

- RVF Methodology Guidelines
  - Functional Coverage
  - Error Injection
- Code Templates
- Training Materials with labs
- Examples
  - Ethernet data model
  - Functional Coverage
  - Error injection
  - VIP wrapping

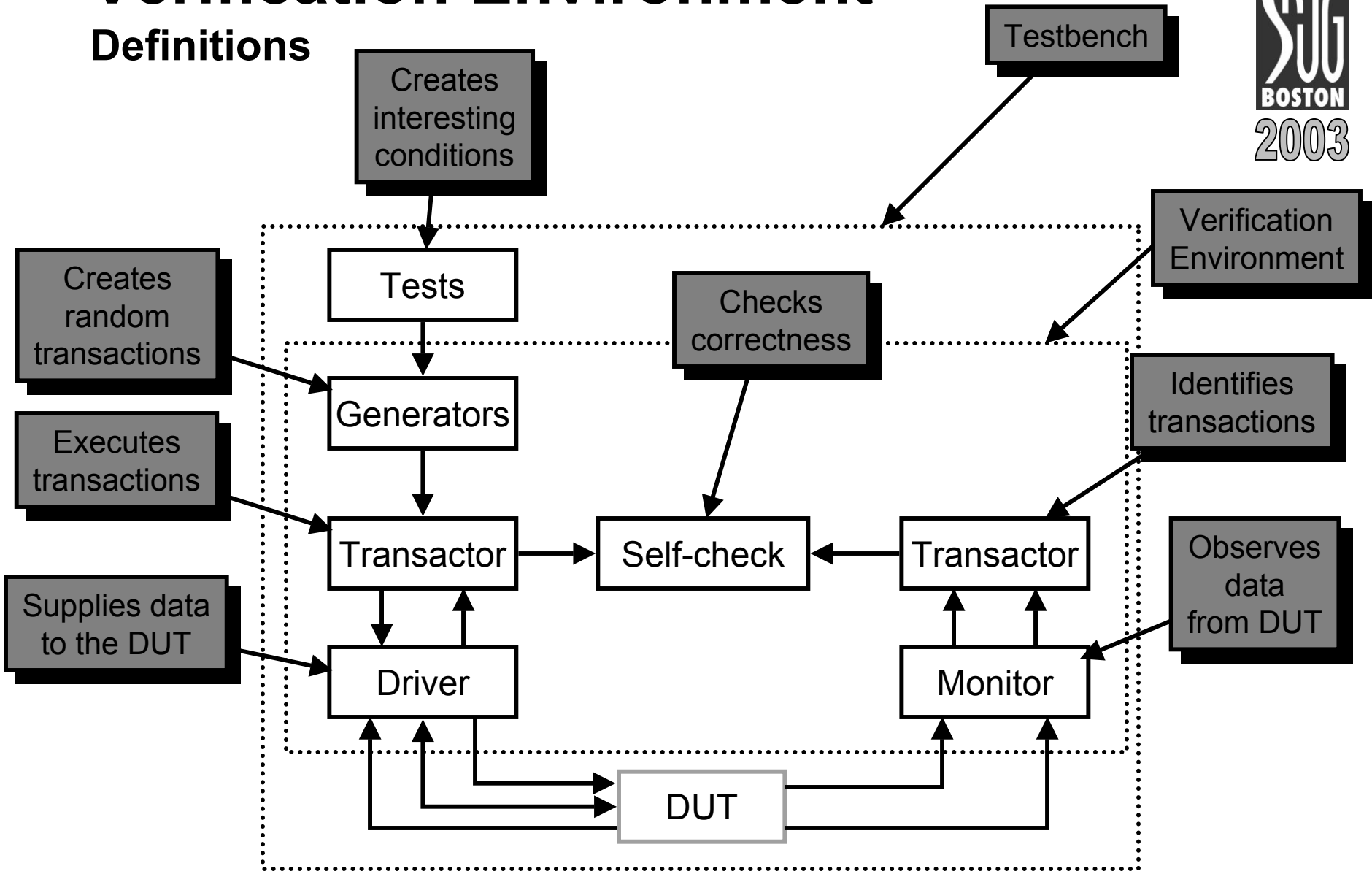
- Common message service
  - VERA
  - OVA
  - VIP
- Automation
  - Face builder

Outline

- Verification Environment
- Data and Transactions
- Transactors
- Generators
- SystemVerilog

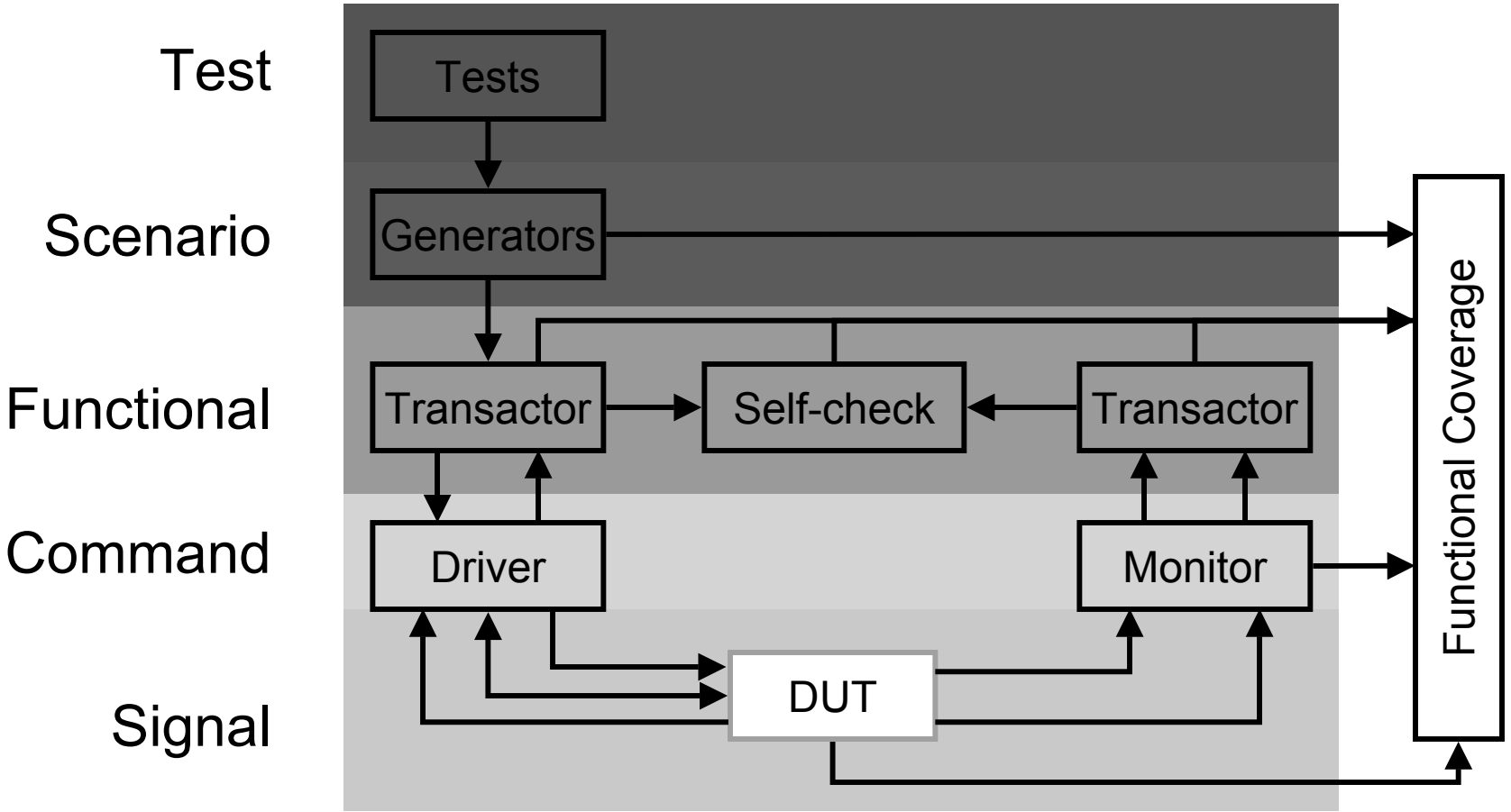
Verification Environment

Definitions



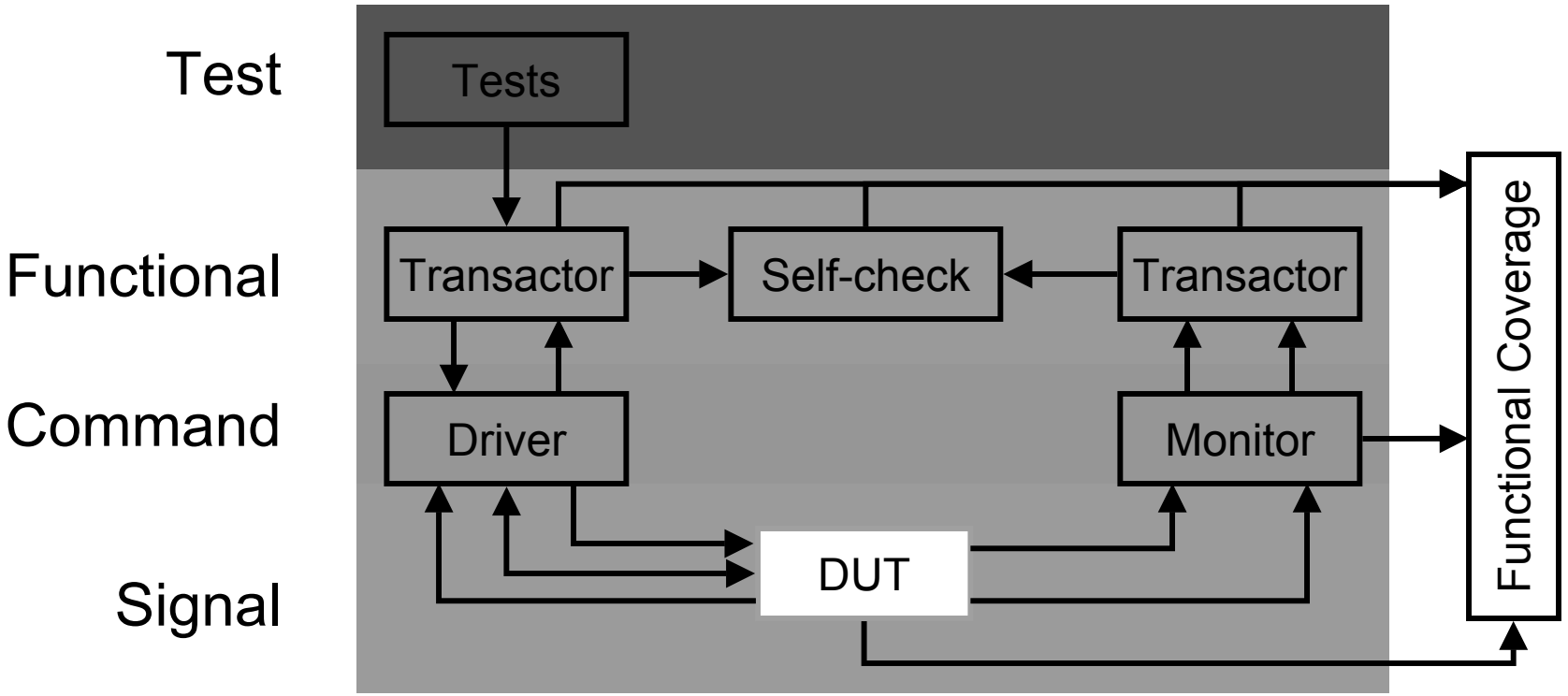
Architecture
Layers

- Constrained random tests



Architecture
Layers

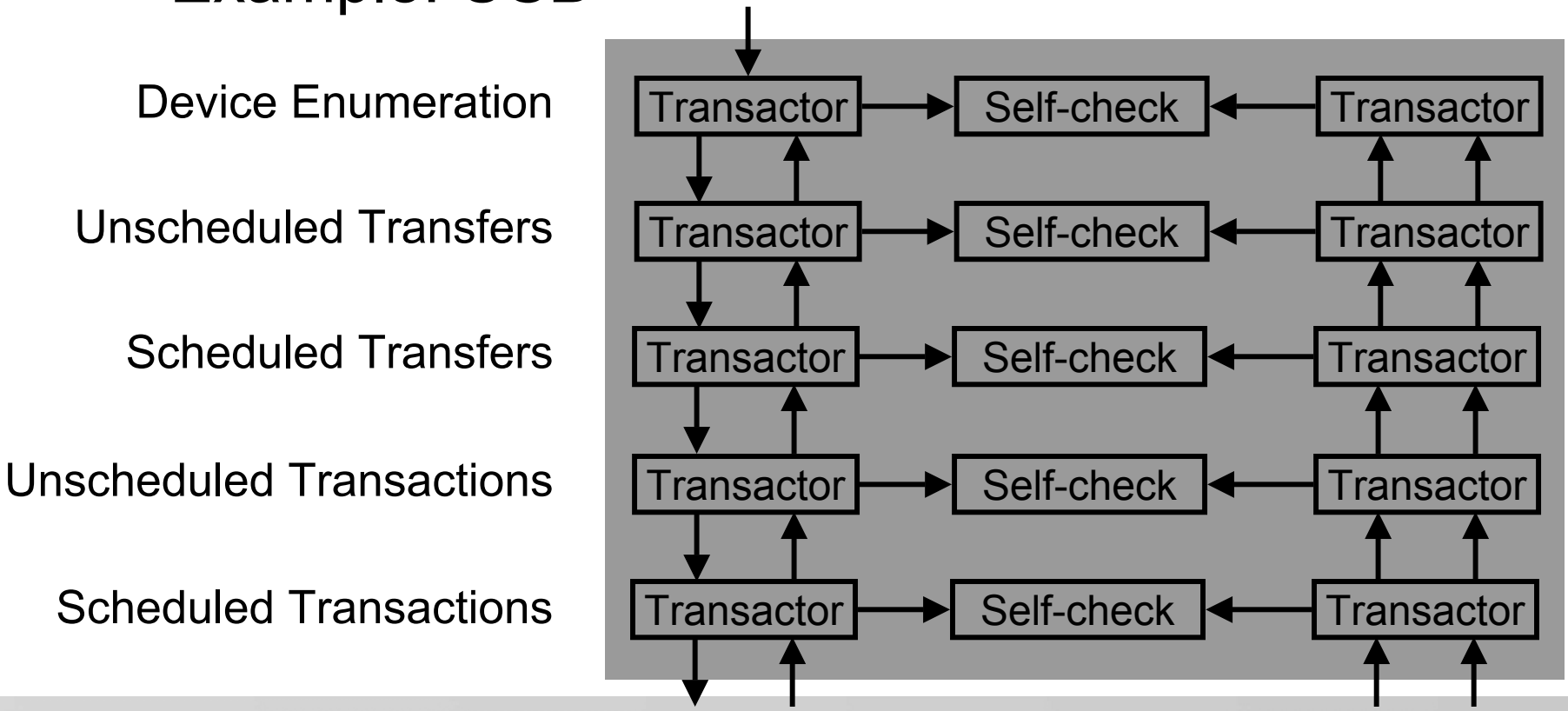
- Directed tests



Functional Layer
Overview

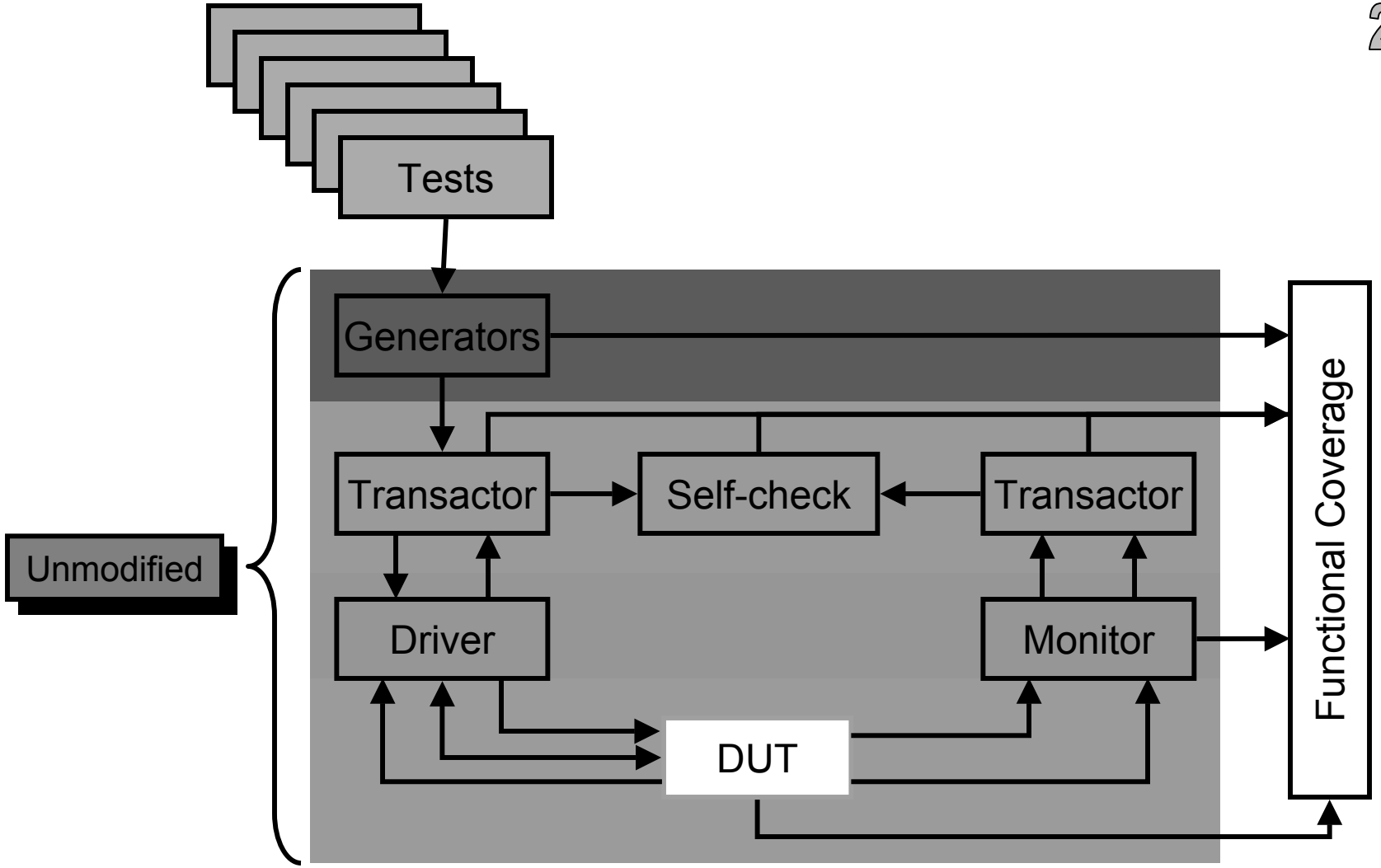
- May be divided into sub-layers
  - According to protocol layers
- Example: USB

As high-level
As required



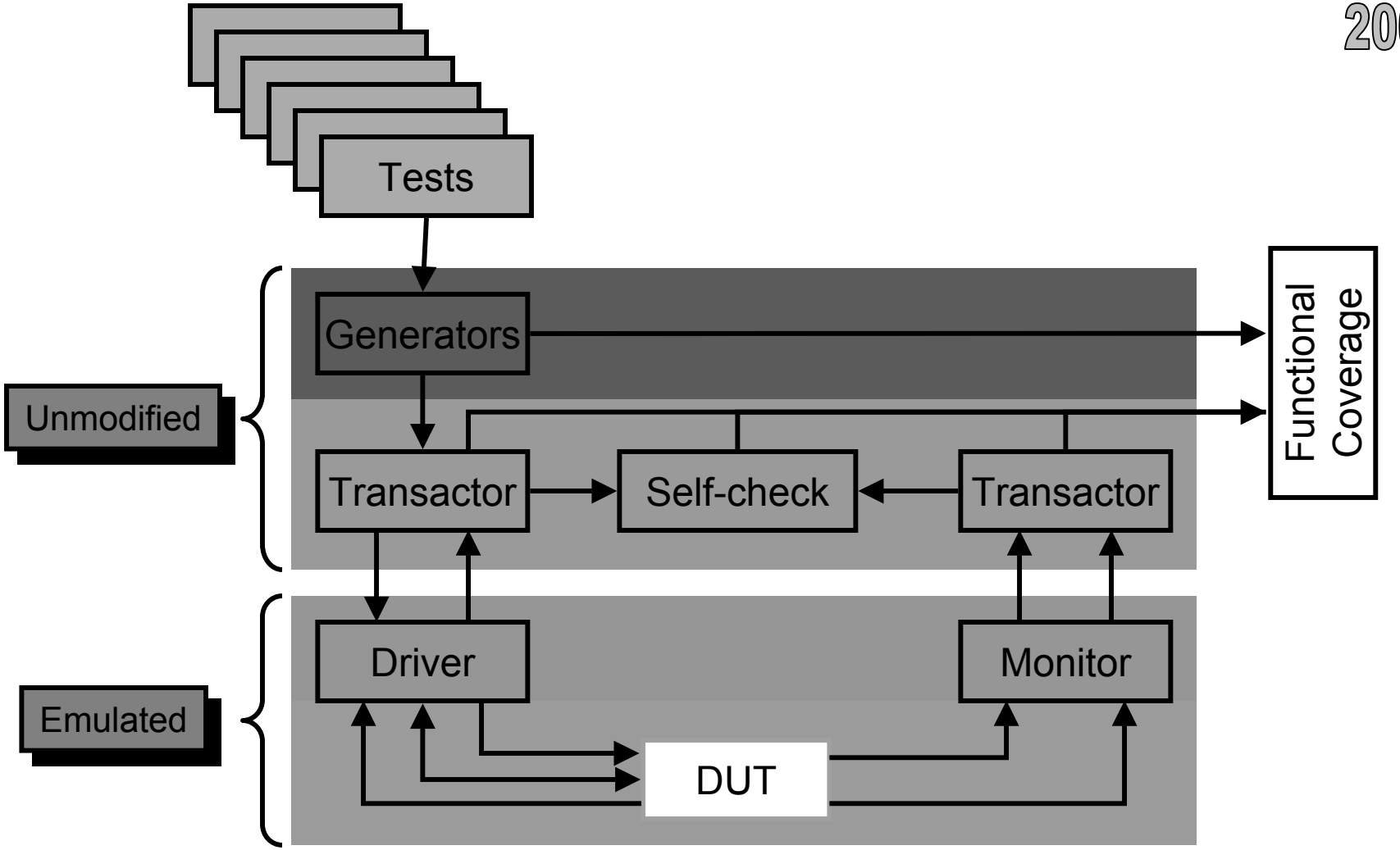
Verification Reuse

Across Testcases



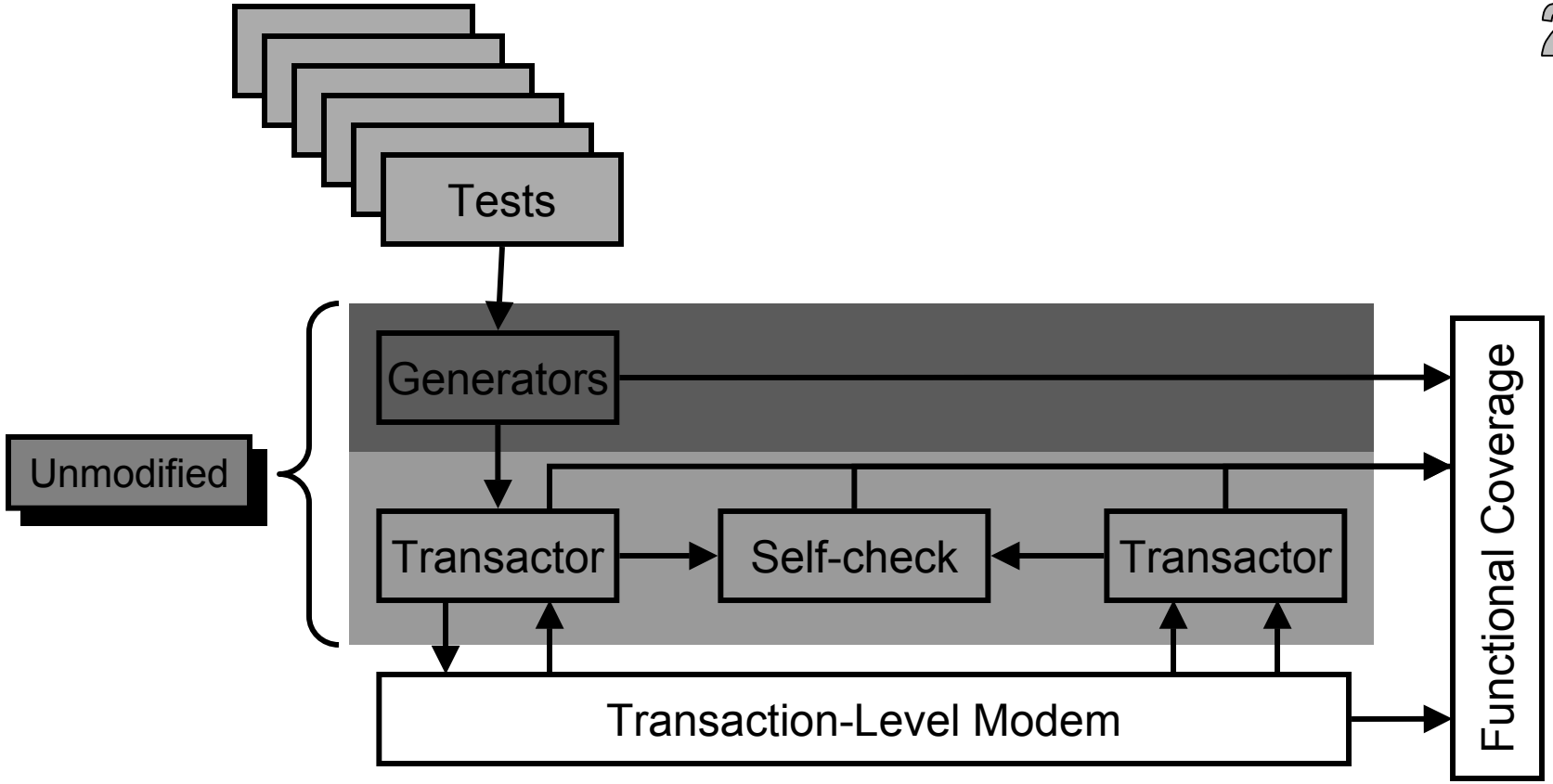
Verification Reuse

Across Simulators



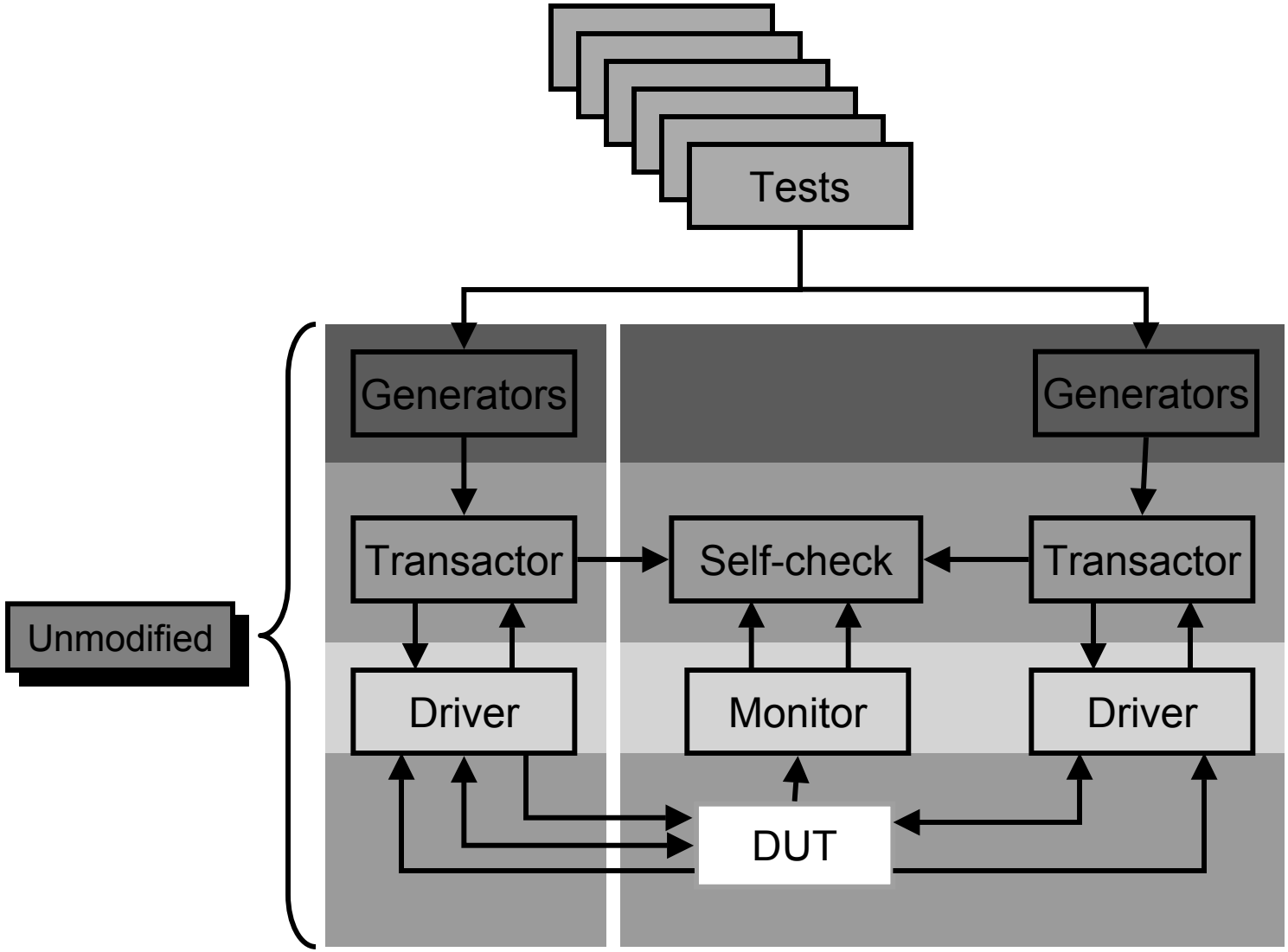
Verification Reuse

Across Models



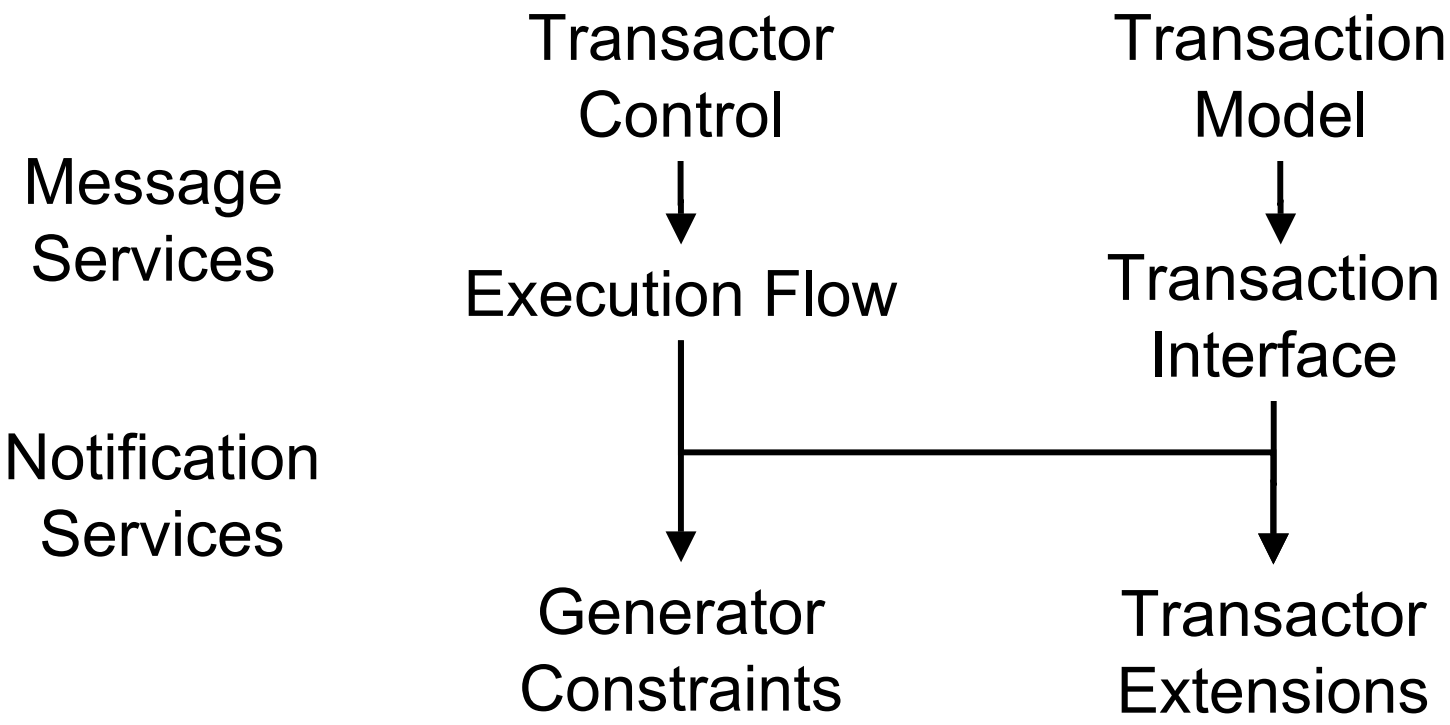
Verification Reuse

Across Projects



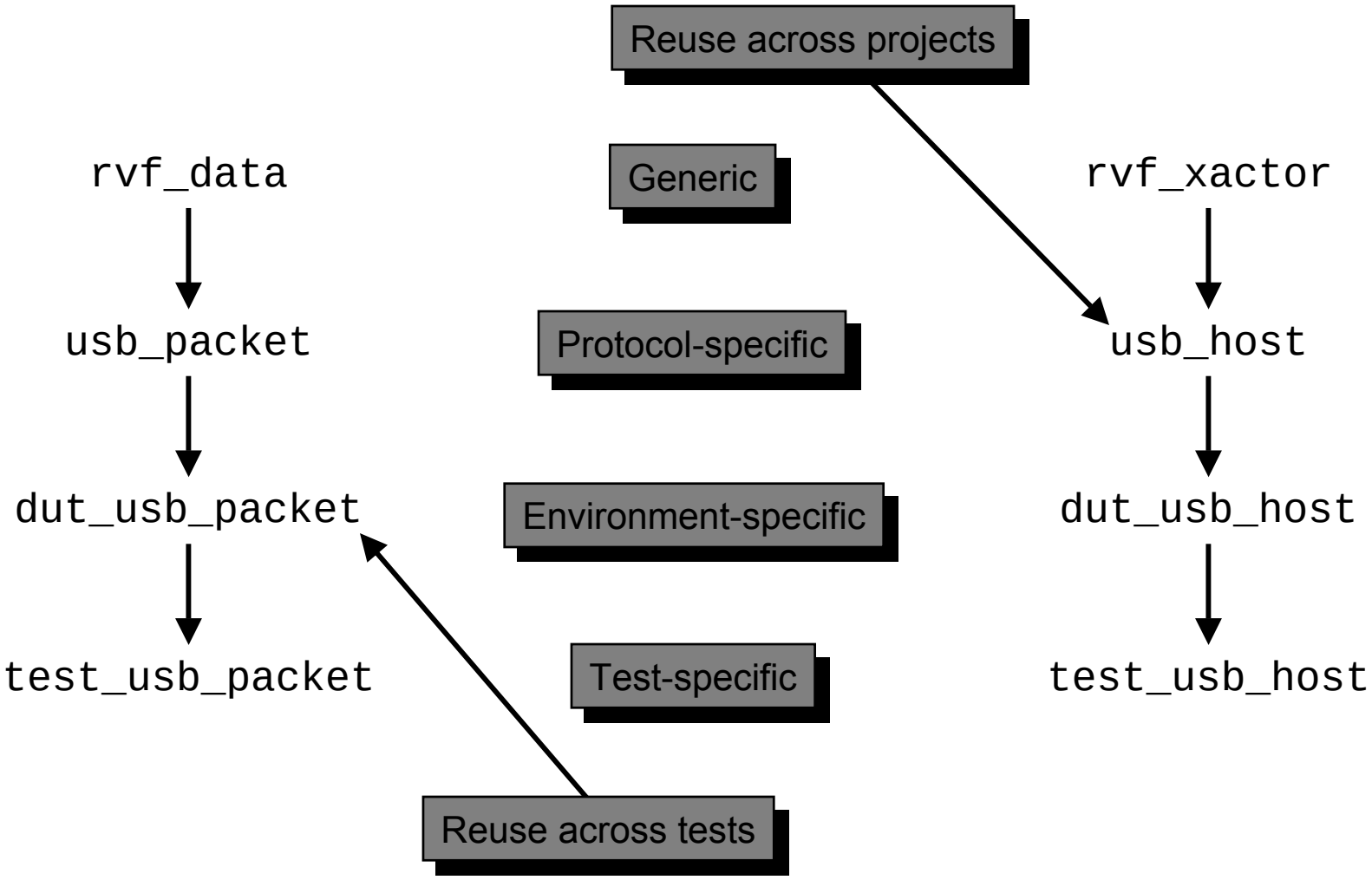
Reference Verification Flow

Synergies



Verification Reuse

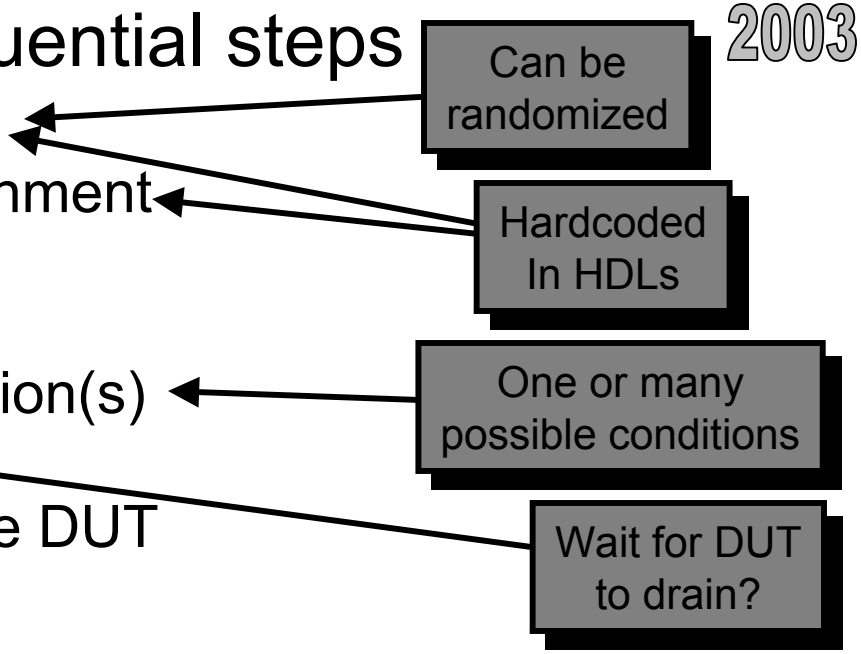
Class Hierarchy



Execution Flow

Steps

- Simulation involves sequential steps
  - Generate test configuration
  - Build the verification environment
  - Configure the DUT
  - Start the flow of data
  - Wait until end-of-test condition(s)
  - Stop the flow of data
  - Download statistics from the DUT
  - Check of lost data
  - Declare pass/fail

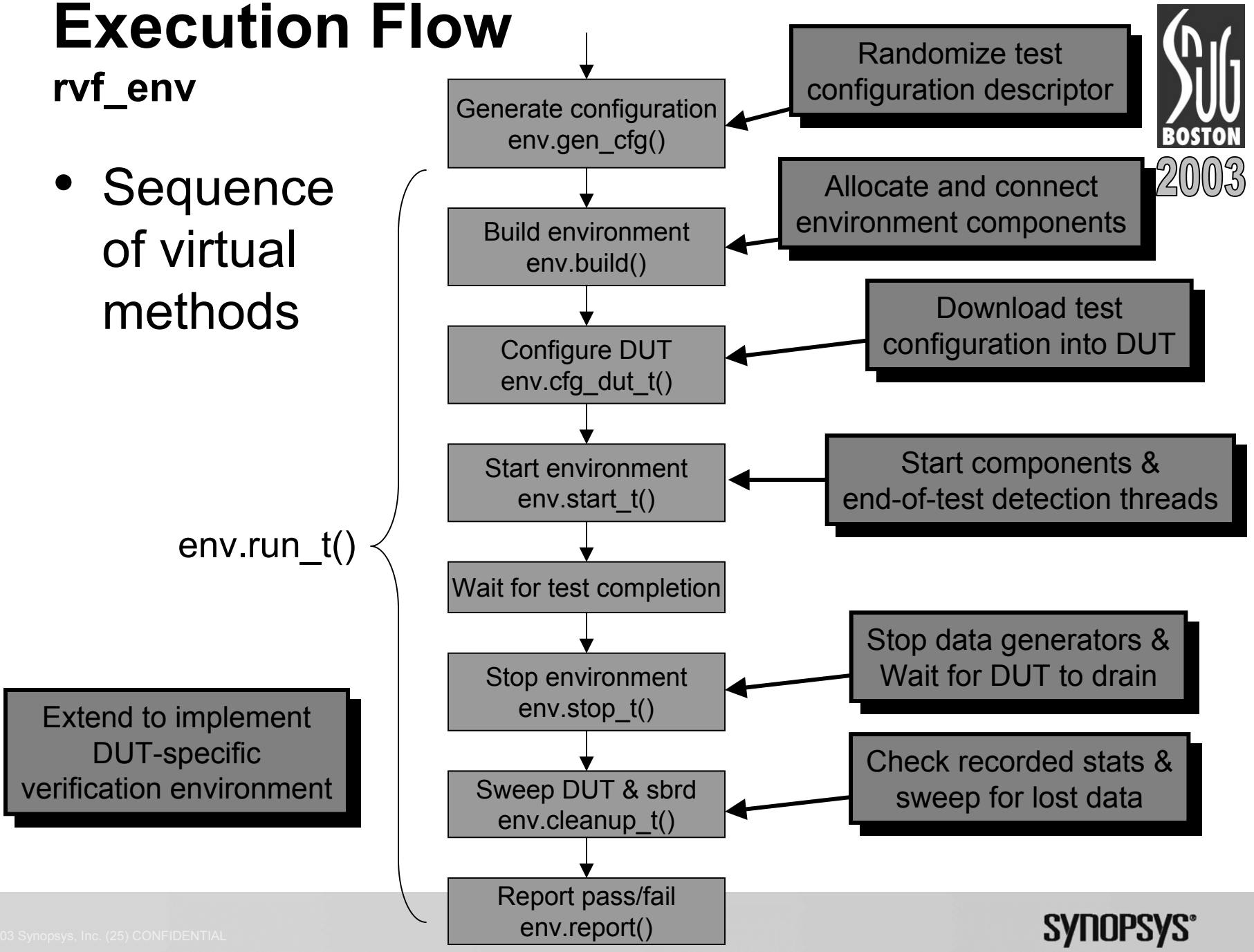


- Different tests need to intervene at different steps
- Base class: *rvf_env*

Execution Flow

rvf_env

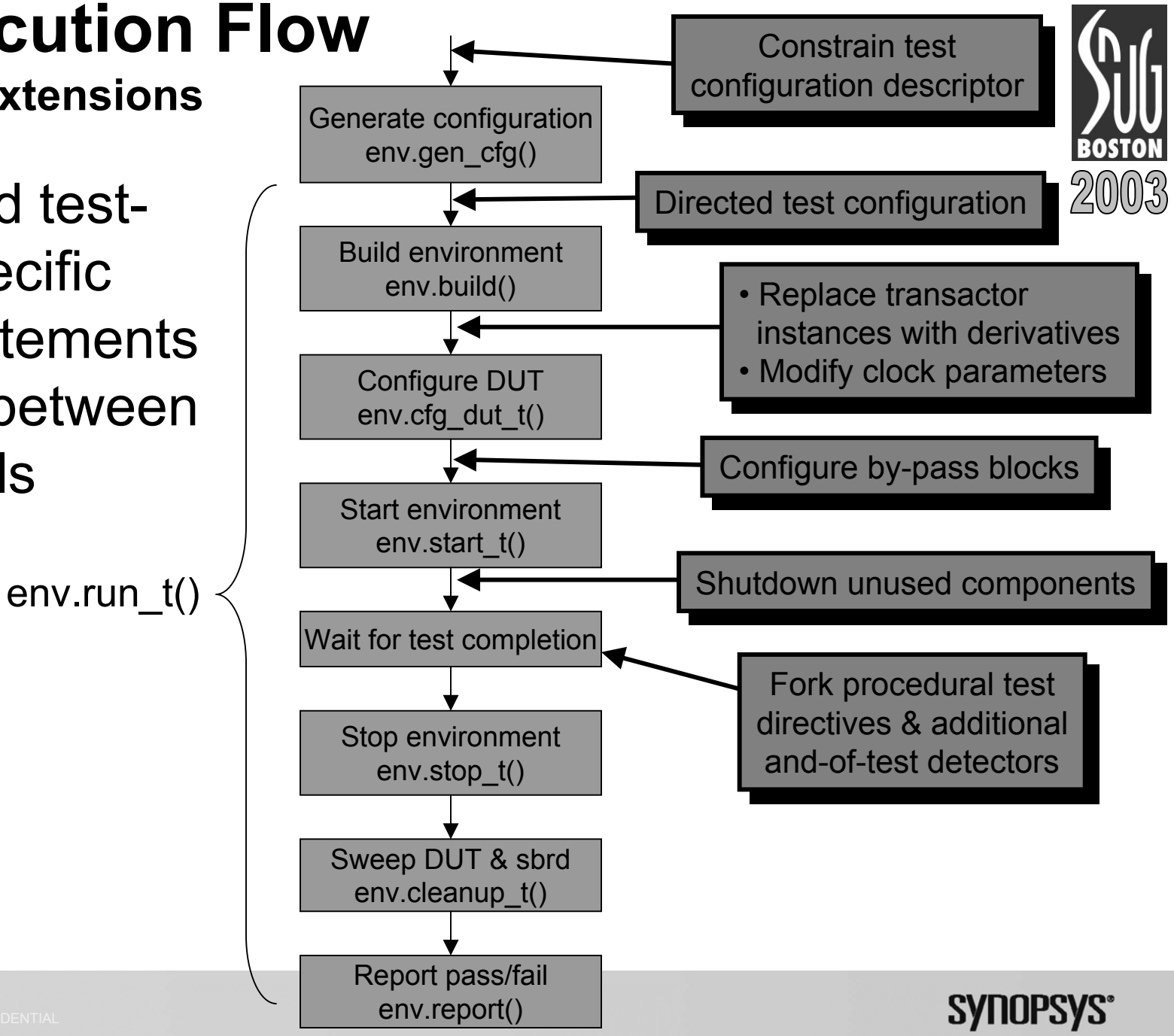
- Sequence of virtual methods



Execution Flow

Test extensions

- Add test-specific statements in-between calls



Execution Flow

Test extensions

- Simplest test

```
program test {
   verif_env env = new;

   env.run_t();
}
```

- Initial debug, trivial test

```
program test {
   verif_env env = new;

   env.gen_cfg();
   env.randomized_cfg.run_for_n_frames = 1;
   env.run_t();
}
```

Execution Flow

Test extensions

- Directed test

```
program test {
   verif_env env = new;

   env.gen_cfg();
   env.randomized_cfg.run_for_n_frames = 999;
   env.randomized_cfg.in_use = 4'b1111;
   env.start_t();
   for (i = 0; i < 4; i++) {
      env.src[i].stop();
   }
   fork
      {
         ...
         trigger(env.sb.enough);
      }
   join none
   env.run_t();
}
```

Execution Flow

Test extensions

- Corner-case test

```
class my_eth_frame extends eth_frame {
   bit [47:0] mac_address;
   constraint one_port_only {
      da == mac_address;
   }
}

program test {
   verif_env env = new;

   env.gen_cfg();
   env.randomized_cfg.in_use = 4'b0001;
   env.build();
   {
      my_eth_frame my_fr = new;
      my_fr.mac_address = env.randomized_cfg.mac_address[0];
      env.src[0].randomized_fr = my_fr;
   }
   env.run_t();
}
```

Messages
rvf_log class

- Pre-defined message service class
- User-defined message format
- Each instance independently controlled
  - Can be controlled via any other instance
  - Controlled via name and instance name
    - Via regular expressions
    - E.g. "all messages issued by instances of the component named 'AHB Master' with instance names containing the string "0" will issue debug-level trace messages"
  - Controlled recursively
    - E.g. "All components instantiated in the AHB sub-system of the verification environment will not issue any trace messages"

rvf_log class
Constructor

- Syntax

```
task rvf_log::new(string name,
                  string instance);
```

- Name
  - Name of the object containing the *rvf_log* instance
  - Describes nature of the object
  - Independent of instance name
  - Example: "AHB Master", "USB Host", "MAC Frame"
- Instance Name
  - Name of the object instance
  - Describes the instance of the object
  - Independent of name
  - Example: "U0", "Stream #0", "Left side"

rvf_log class
Issuing Simple Messages

- Syntax

```
rvf_warning(rvf_log log,
            string msg);
```

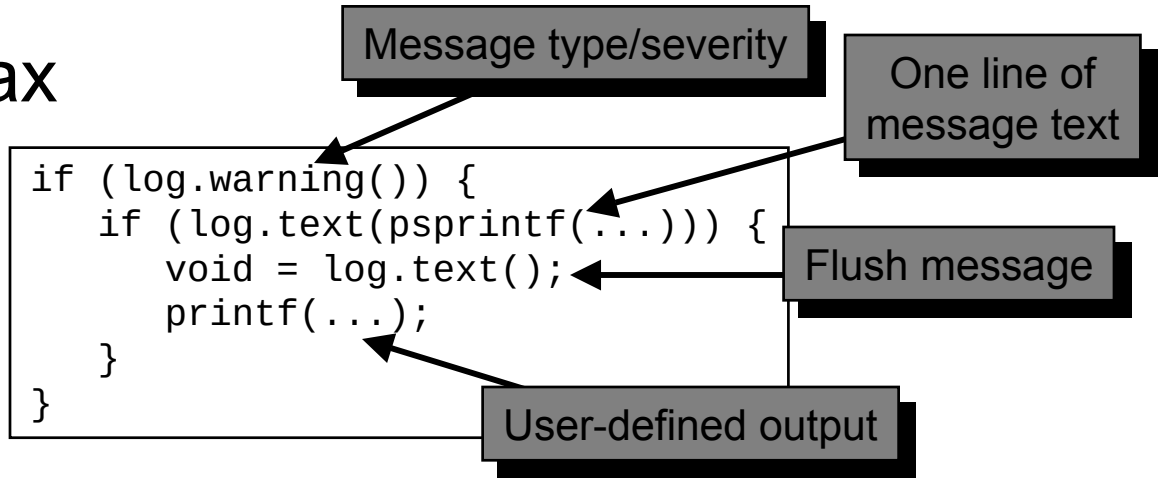
- For single-line messages
- Issued via the specified *rvf_log* instance
- Other macros

```
rvf_error(rvf_log log, string msg);
rvf_fatal(rvf_log log, string msg);
rvf_note(rvf_log log, string msg);
rvf_trace(rvf_log log, string msg);
rvf_debug(rvf_log log, string msg);
rvf_verbose(rvf_log log, string msg);
```

rvf_log class

Issuing Simple Messages with User-Defined Output

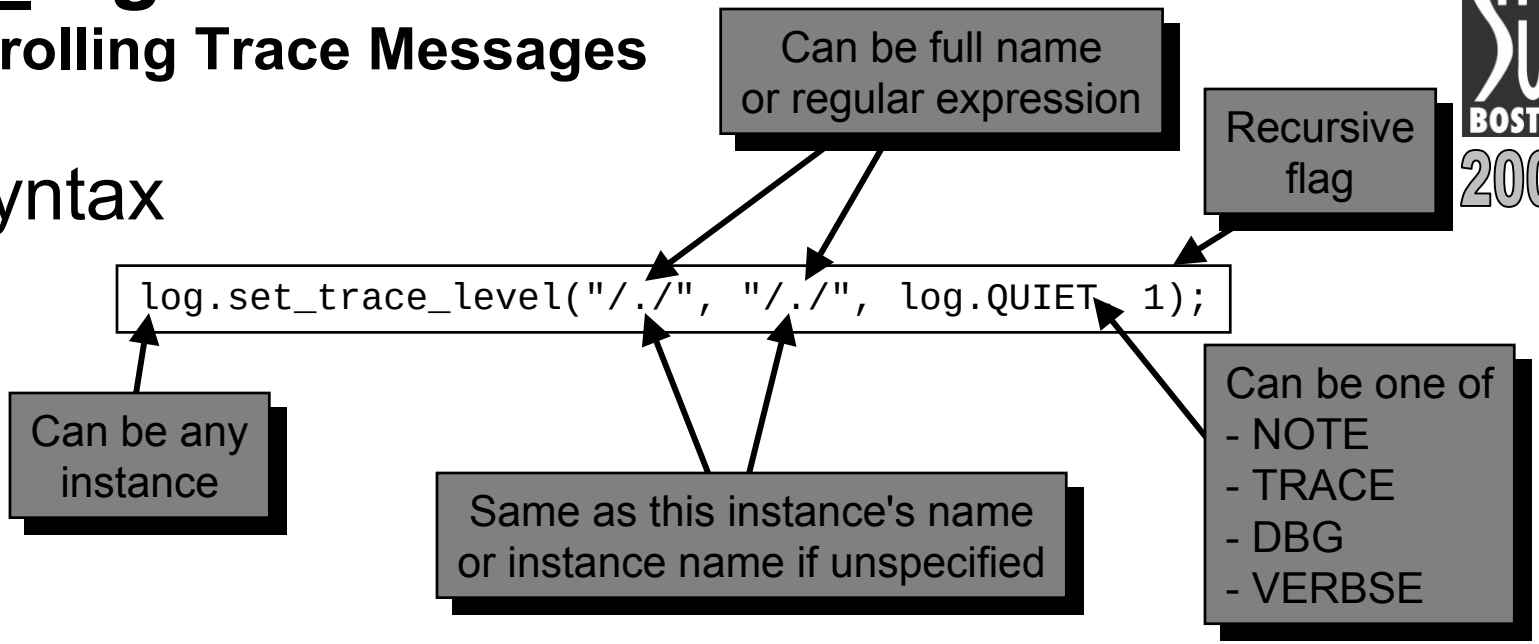
- Syntax



- For single-line messages followed by user-defined output and *printf()* statements
- Other methods

```
function bit rvf_log::error();
function bit rvf_log::fatal();
function bit rvf_log::note();
function bit rvf_log::trace();
function bit rvf_log::debug();
function bit rvf_log::verbose();
```

rvf_log class
Controlling Trace Messages

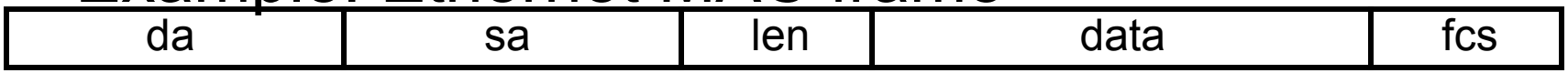


- Syntax
- Can control any and all *rvf_log* instances for any instance
  - By name
  - Recursively
- To specify this instance only:

```
log.set_trace_level(*, *, log.DBG);
```

Data Models
Data Properties

- Every data field in specification modeled as a randomized public data properties
  - Don't hide as *local* properties with *get()*/*set()*methods
  - Must be visible to constrain in
    - Derived class
    - *Randomize() with* construct
    - Scenario descriptors
- Example: Ethernet MAC frame



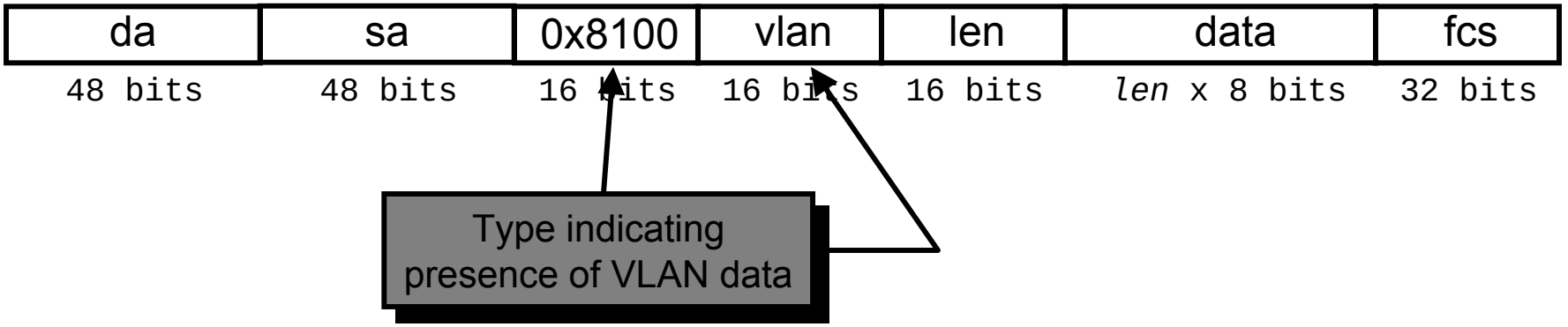

```
class eth_mac_frame extends rvf_data {
   rand bit [47:0] da;
   rand bit [47:0] sa;
   rand bit [15:0] len;
   rand bit [ 7:0] data[$];
   rand bit [31:0] fcs;
}
```

Variant Data
Variant Fields

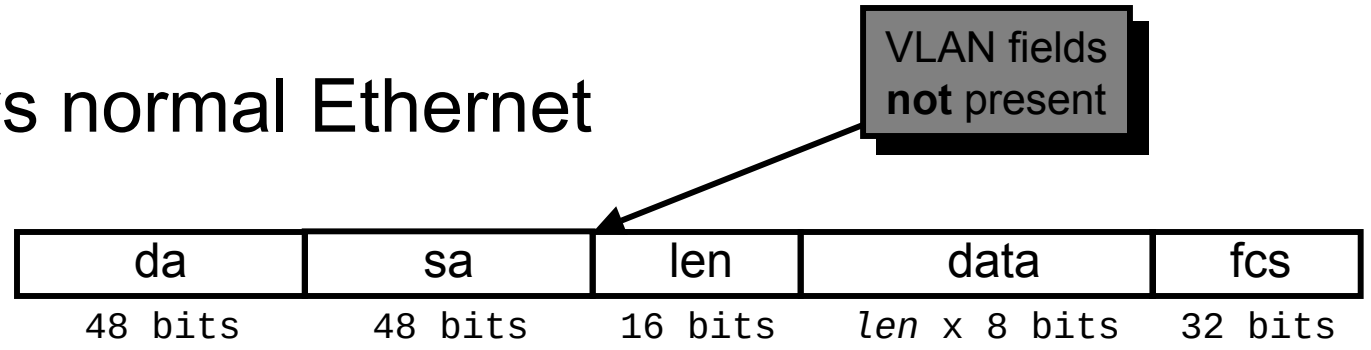
- Different fields may be available under different cases
- Use property to define case ("discriminant")
  - Declare property *rand*
- Define all fields in same class
- Conditionally use properties based on discriminant property

Variant Data
Example

- Example: Virtual LAN in Ethernet



- vs normal Ethernet



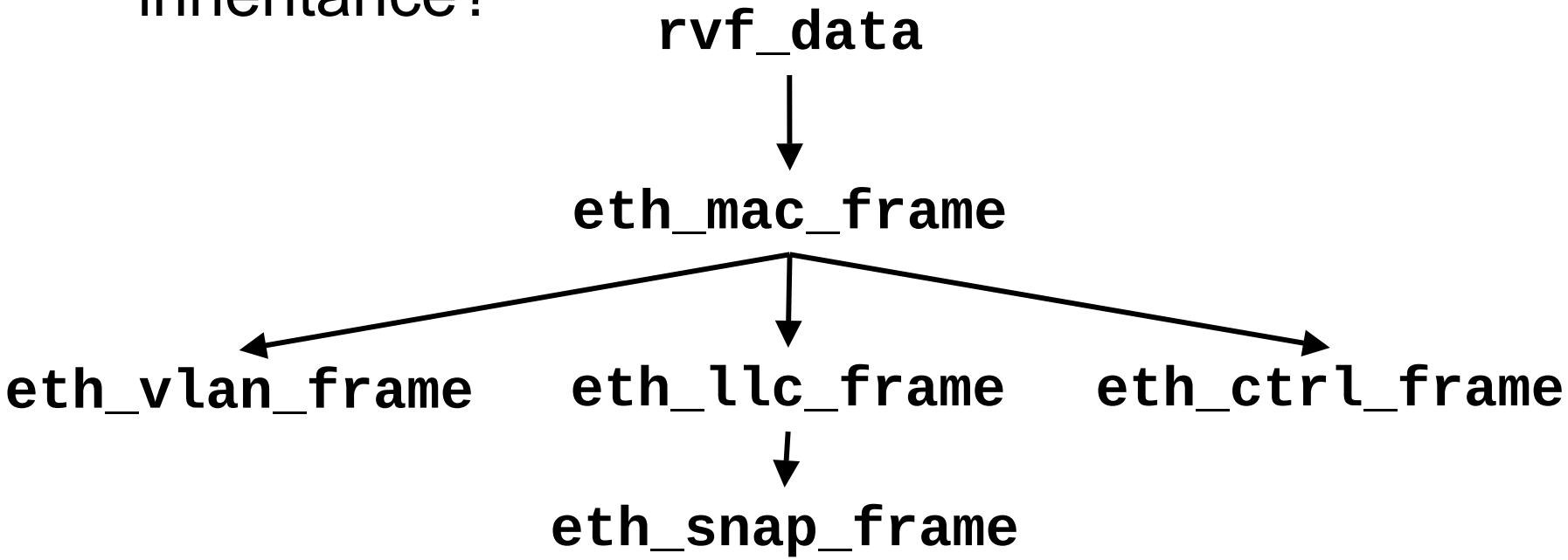
Variant Data
Example


```
class eth_mac_frame extends rvf_data {
   ...
   rand bit has_vlan;
   rand bit [ 2:0] priority;
   rand bit        cfi;
   rand bit [11:0] vlan_id;
   ...
   virtual function integer byte_pack(var bit [7:0] bytes [*],
                                      integer       offset = 0) {
      ...
      if (has_vlan) {
         `{bytes[i+1], bytes[i]} = 0x8100;                    i += 2;
         `{bytes[i+1], bytes[i]} = {priority, cfi, vlan_id};  i += 2;
      }
      ...
   }
}
```

Variant Data

Do Not Use Inheritance

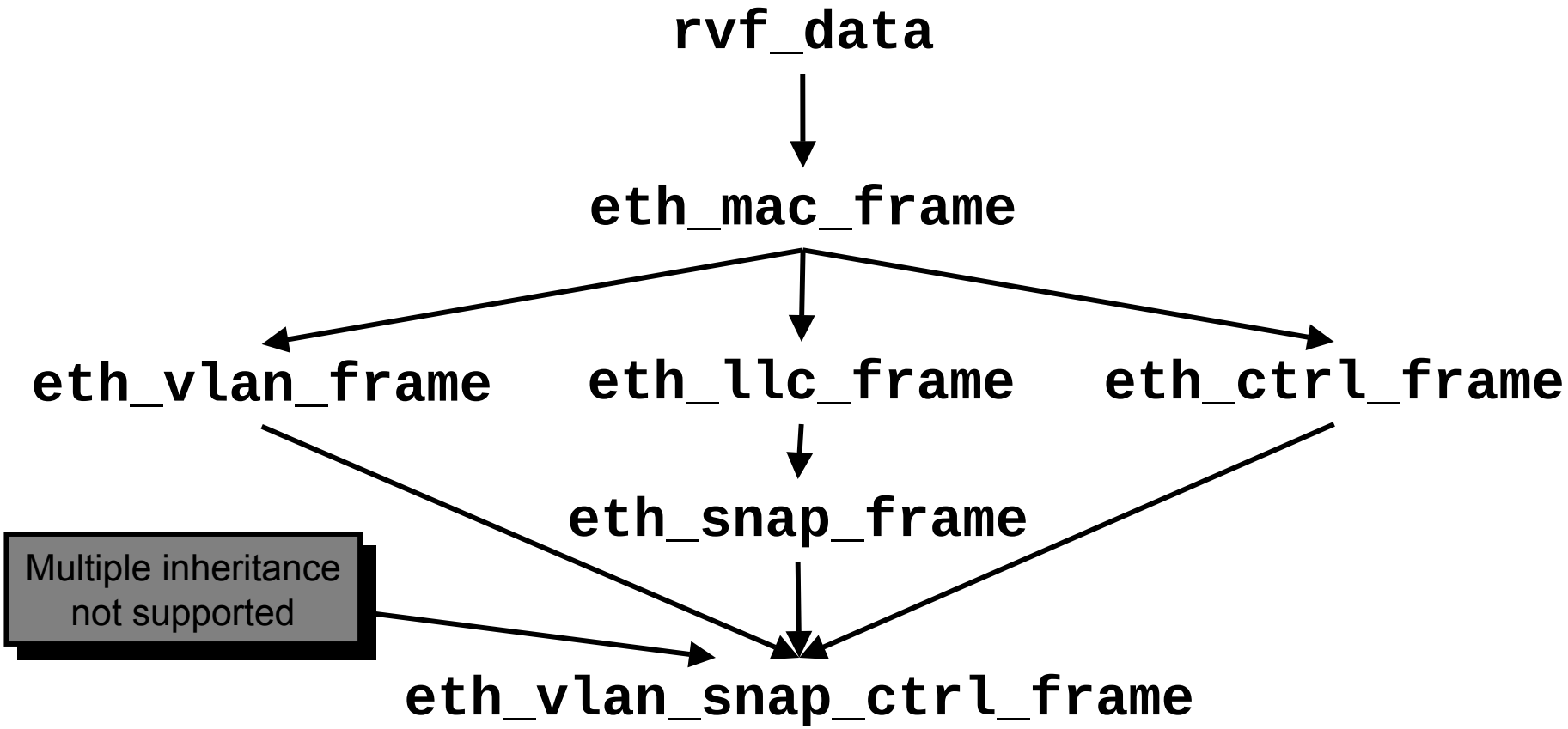
- Why not model variant data using inheritance?



Variant Data

Do Not Use Inheritance

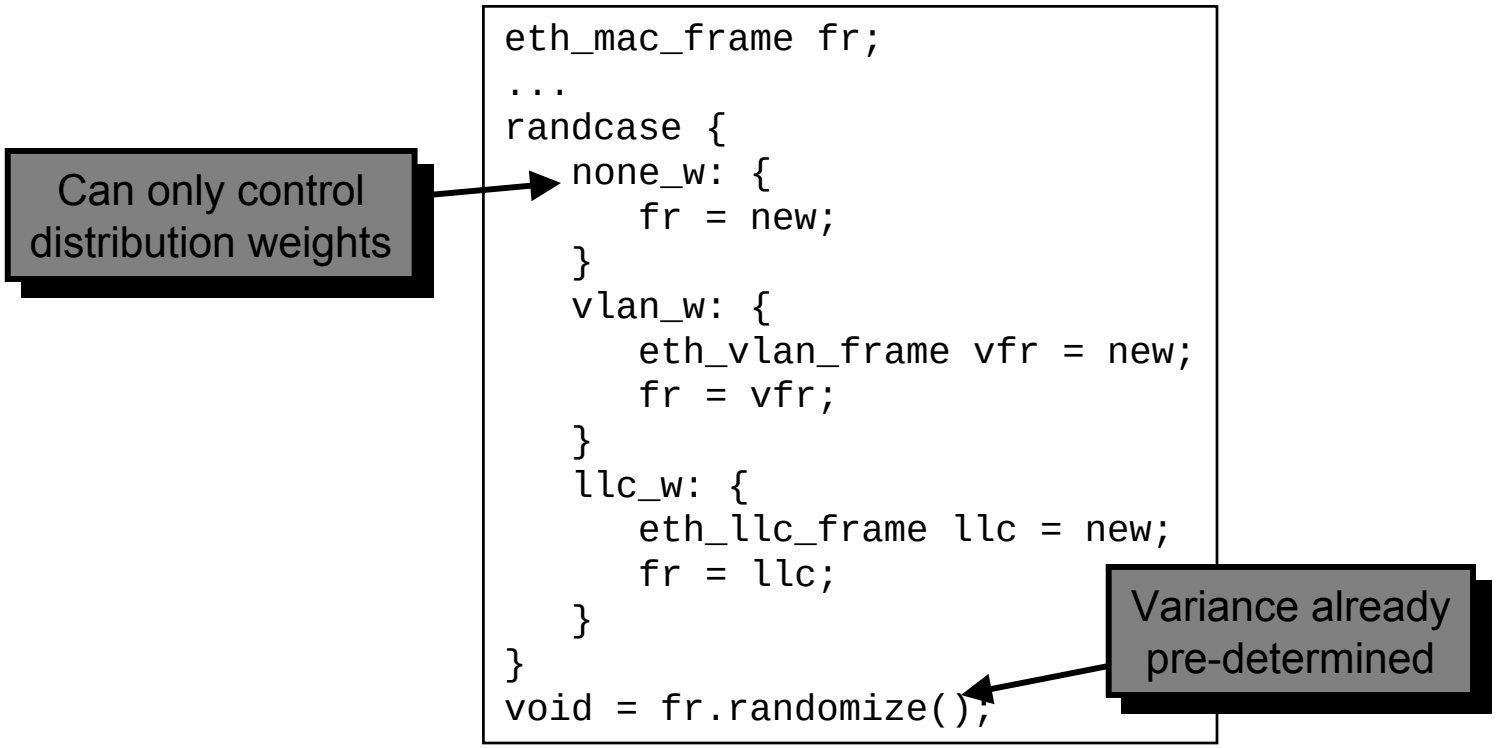
- Cannot mix and match orthogonal variances



Variant Data

Do Not Use Inheritance

- Cannot easily create stream of random variance mix

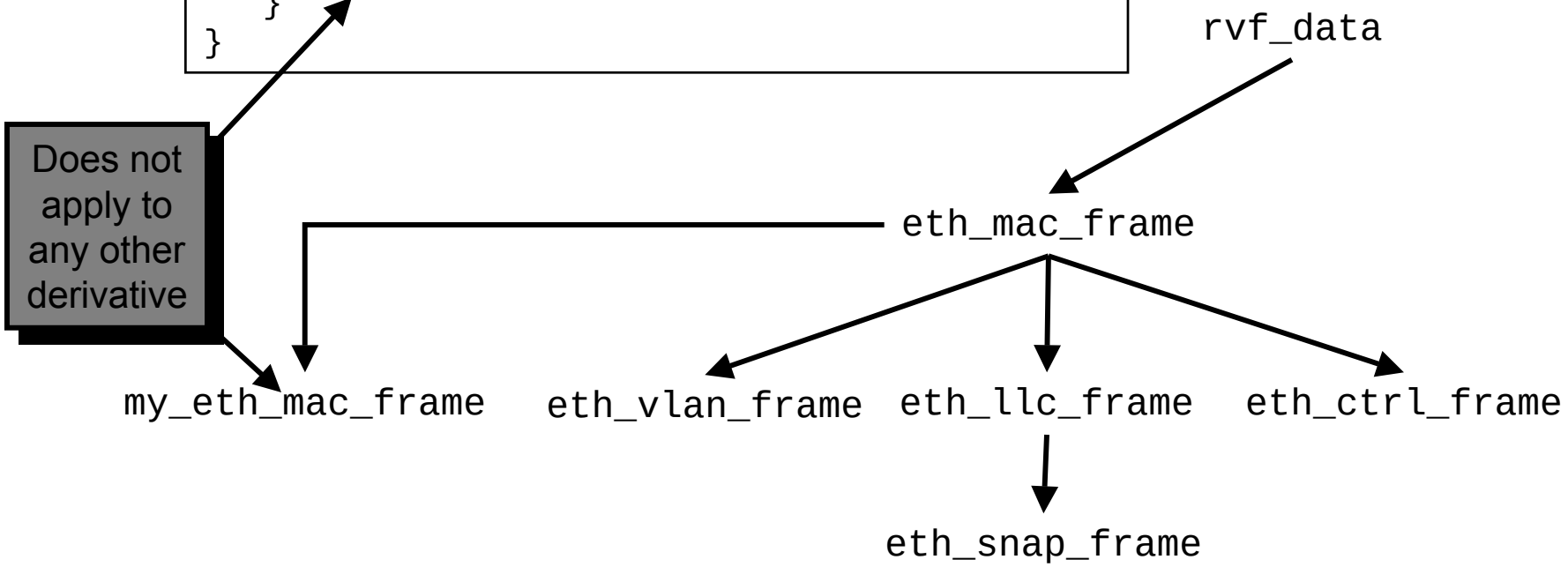


Variant Data

Do Not Use Inheritance

- Impossible to add shared constraints to common properties

```
class my_eth_mac_frame extends eth_mac_frame {
   constraint small_packets {
      data.size() == 42;
   }
}
```



Data Objects in RVF

Standard Methods in *rvf_data*

- All methods are virtual
- Must be defined in all extensions

```
class rvf_data {
   ...
   task display(string prefix = "");

   function rvf_data allocate();
   function rvf_data copy(rvf_data to = null);

   function bit compare(rvf_data to);

   function integer byte_pack(var bit [7:0] bytes[*],
                              integer offset = 0,
                              integer kind   = -1);
   function integer byte_unpack(bit [7:0] bytes[*],
                                integer offset = 0,
                                integer kind   = -1);
   function integer byte_size();

}
```

Transactions
Procedural Interface

- Transactions are usually modeled as procedures
  - One procedure per transaction

```
class apb_master {
   ...
   virtual function bit [31:0] read(bit [ 7:0] sel,
                                    bit [31:0] addr);
   virtual task write(bit [ 7:0] sel,
                      bit [31:0] addr,
                      bit [31:0] data);
}
```

Transactions

Transactions as Objects

- Model transactions as objects
  - Individual transactions are variants
- Transactor interpret object to execute transaction

```
class apb_transaction {
   enum kind_t = READ, WRITE;
   rand kind_t kind;

   rand bit [ 7:0] sel;
   rand bit [31:0] addr;
   rand bit [31:0] data;
}
```

```
class apb_master {
   task do(apb_transaction tr) {
      case (tr.kind) {
         ahb_transaction::READ: {
            ...
         }
         ahb_transaction::WRITE: {
            ...
         }
      }
   }
}
```

Transactions
Generating Transactions

- Simple to generate stream of random transactions

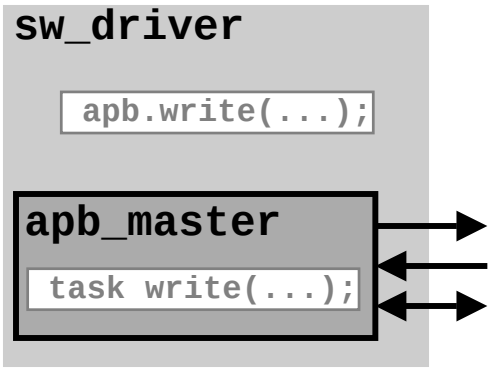
```
apb_transaction tr = new;
void = tr.randomize();
apb.do(tr);
```

- Can use OOP constraints and solver
  - Add or override constraints via extensions
  - Constraints across multiple instances

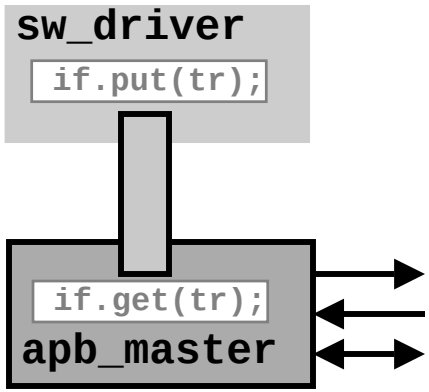
```
class my_apb_tr extenda apb_ansaction {
   constraint available_targets {
      sel in {0:3};
      if (sel == 0 && kind == READ) {
         addr in {0:15, 27, 31};
      } else {
         addr < 31;
      }
   }
}
```

Transactions
Transaction Interface

- Can use transaction interface object
  - Transactors connected via objects



Procedural interface

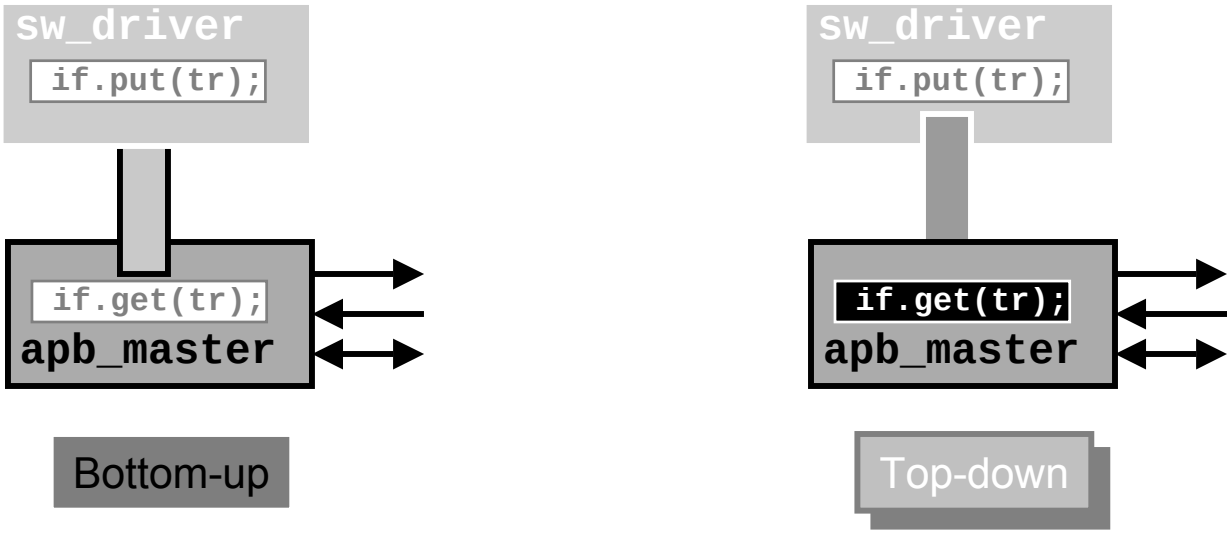


Interface object

Transactions

Transaction Interface Object

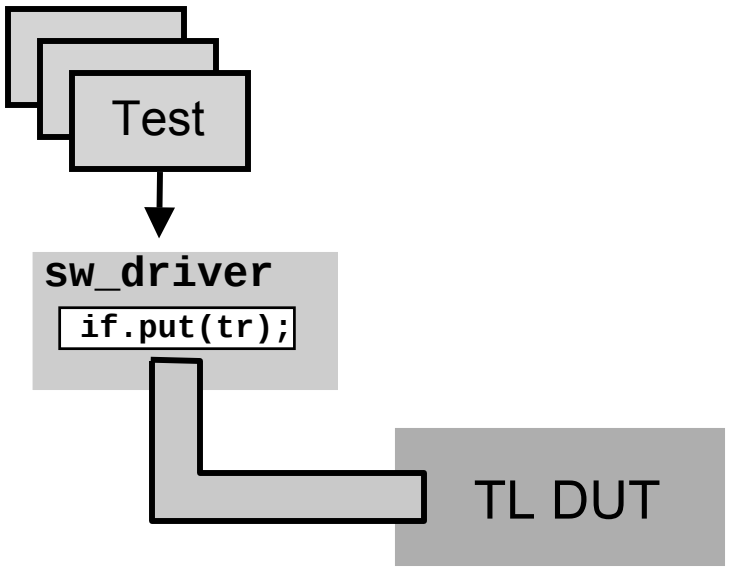
- Can be assembled in any order

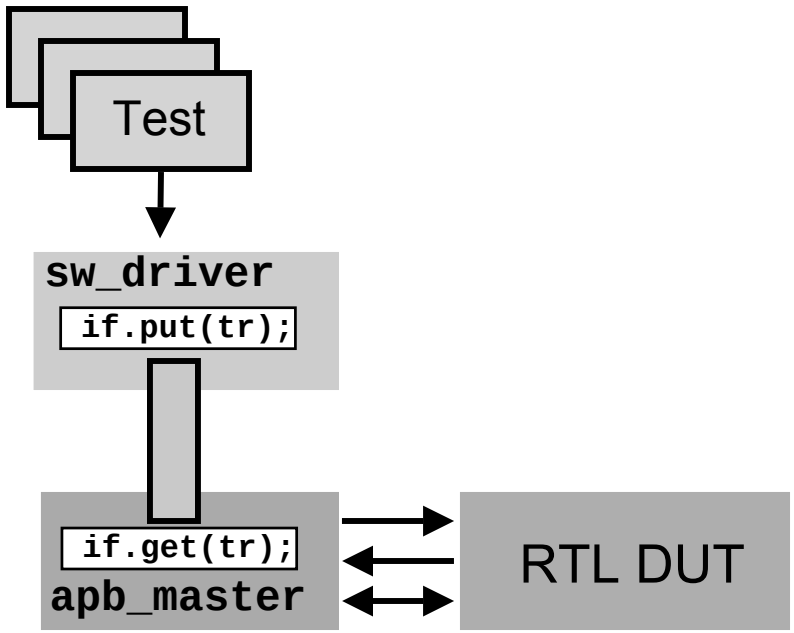


Transactions

Transaction-Level Models

- Can reuse verification environment from transaction-level model

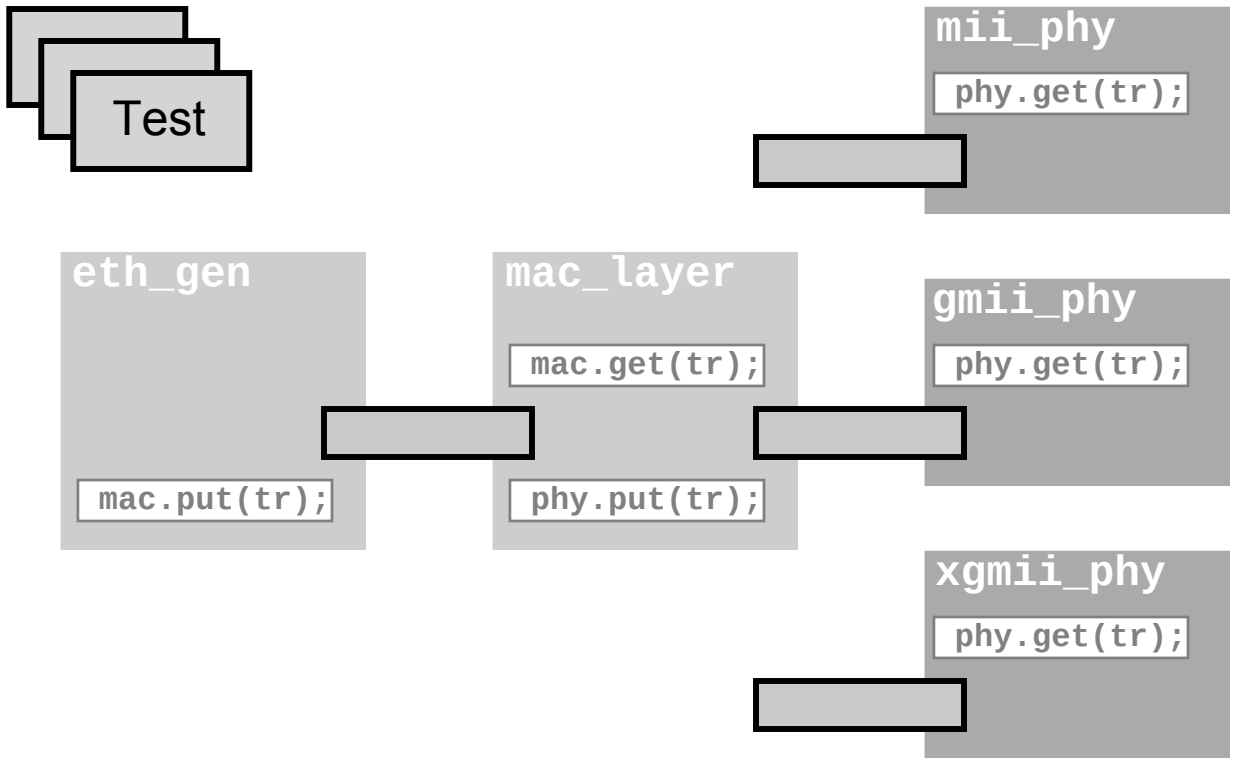




Transactions

Transaction Interface Objects

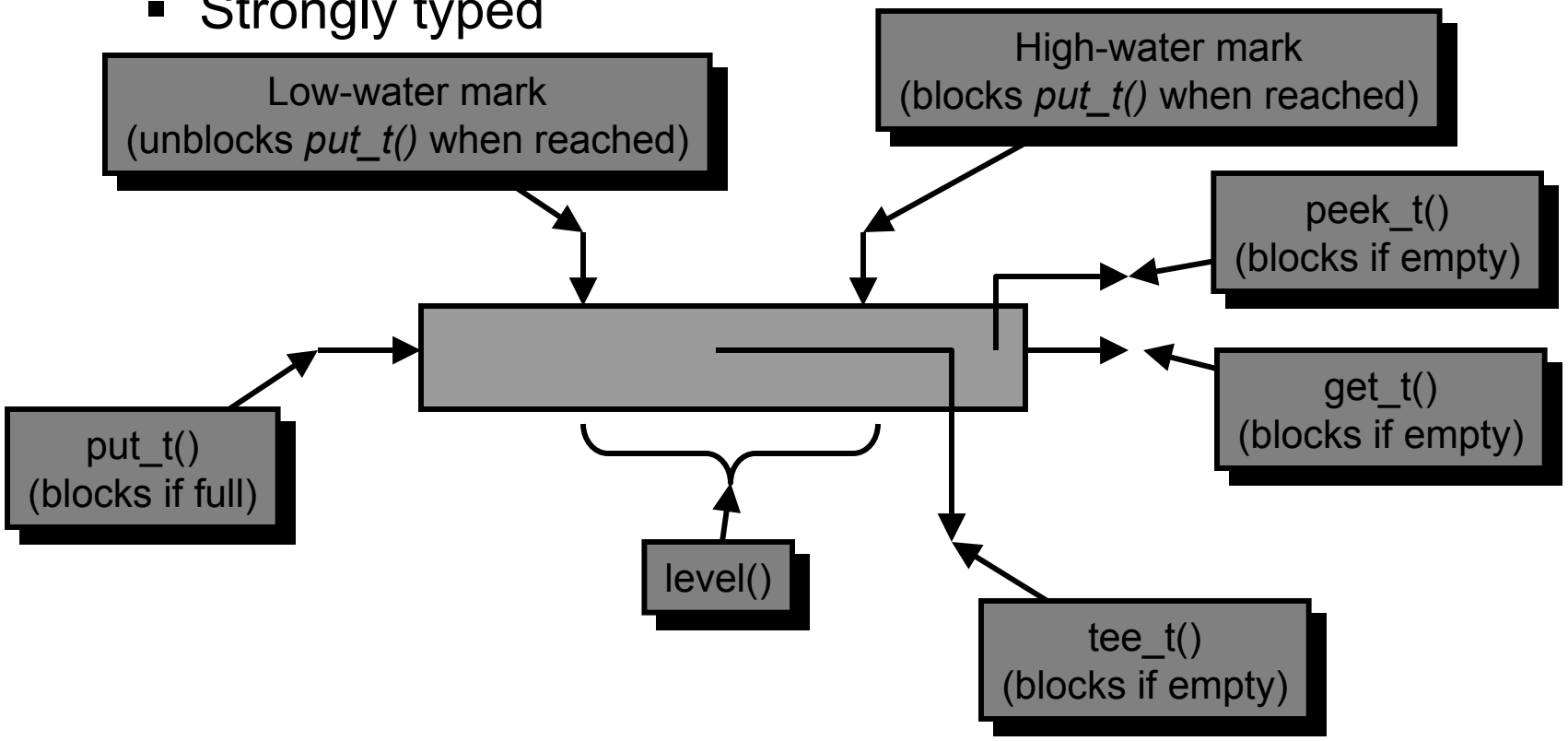
- Can mix and match transactors without modifications



Transaction Interface Object

rvf_channel class

- Transaction interface class
  - Mailbox + control flow
  - Strongly typed



Transaction Interface Object
rvf_channel class

- Can only carry *rvf_data* derivatives
- Declared using macro
  - Defines new class
  - Similar to MakeVeraList

```
class eth_mac_frame extends rvf_data {
   ...
}
rvf_channel(eth_mac_frame)
...
eth_mac_frame_channel if = new(...);
fork
   while (1) {
      eth_mac_frame fr = new;
      if.put_t(fr);
   }
   while (1) {
      eth_mac_frame fr = if.get_t();
   }
join
```

No ';'

Both threads now self-regulated

Transactors
Synchronization

- Use *void* drive and expects to synchronize to interface clocks
  - Do not wait for explicit edge on clock signals
    - Active edge can be modified in *interface* declaration
    - Will break functionality of physical interface
  - Not including clocks in virtual ports prevents this

Don't

```
@ (posedge this.sigs.$txc);
...
@ (posedge this.sigs.$rxc);
```

Do

```
@1 this.sigs.$txd = void;
...
@1 this.sigs.$rxd == void;
```

Reusable Transactors
Callback Methods

- Need mechanism to provide additional services to verification environment
  - Introducing delays
  - Synchronizing different transactors
  - Feedback
  - Recording input into scoreboard
  - Comparing output with expected values
  - Sampling data for functional coverage
  - Modifying transactions to inject errors
  - Deviate from default behavior
- Must be flexible enough to satisfy unpredictable needs of verification environments and testcases

Reusable Transactors
Callback Methods

- Provide rich set of callback methods
  - After a new transaction is about to be started
    - Allow delay insertion
    - Allow modifying or dropping the transaction
  - Before a major decision is about to be made
    - Allow the default decision to be changed
  - Before sub-transactions or data is transmitted
    - Allow delay insertion
    - Allow modifying or dropping
  - After sub-transactions or data has been received
    - Allow modifying or dropping
  - After a transaction has been completed
    - Allow modifying or prevent forwarding to higher layers
  - In the body of every loop
    - Supply loop index via argument
    - Allow modifying the subject of the iteration

If protocol allows it

Reusable Transactors
Callback Methods

- Callback methods are *virtual* methods invoked by the transactor itself
  - Empty by default
  - Always use a *task*
- Implement in separate callback façade object
- Provide all necessary state information via arguments
  - Calling transactor instance
  - Document the effect of modifying argument value

```
class eth_mii_callbacks extends rvf_xactor_callbacks {
   virtual task pre_fr_tx_t(eth_mii       xactor,
                            eth_mac_frame fr,
                            var bit       drop)
   {}
}
```

Reusable Transactors

rvf_xactor::register_callback()

- To register callback extension with transactor
  - Can be unregistered later
- Can have multiple registrations
  - Invoked in registration order

```
class my_eth_mii_callbacks extends eth_mii_callbacks {
   virtual task pre_fr_tx_t(eth_mii       xactor,
                            eth_mac_frame fr,
                            var bit       drop) {
      integer n = random() % 10;
      @n xactor.sigs.txd = void;
   }
}

program test {
   verif_env env = new;
   env.build();
   {
      my_eth_mii_callbacks cb = new;
      env.mii[3].register_callback(cb);
   }
   env.run_t();
}
```

Reusable Transactors
Callback Methods

- Called "callback" because they are methods called by low-level code to execute user-specified high-level code.

```
class eth_mii extends rvf_xactor {
   ...
   while (1) {
      eth_mac_frame fr = this.in_chan.g
      bit drop = 0;

      foreach (super.callbacks, i) {
         eth_mii_callbacks cb;
         if (!cast_assign(cb, super.callbacks[i], CHECK)) continue;

         cb.pre_fr_tx_t(this, fr, drop);
      }

      if (drop) continue;
      ...
   }
}
```

Iterate over all registered callbacks

Only deal with relevant instances

Reusable Transactors

Callback Methods Interfaces

- Document if a callback method must be nonblocking
  - Use "_t" postfix if they can be blocking
  - Detect blocking implementations

```
fork
   rvf_fatal(this.log,
      "Implementation of eth_mii_callbacks::post_fr_rx() was blocking");
join none
foreach (super.callbacks, i) {
   eth_mii_callbacks cb;
   if (!cast_assign(cb, super.callbacks[i], CHECK)) continue;

   cb.post_fr_rx(this, fr, drop);
}
terminate;
if (!drop) this.to_mac.put_t(fr);
```

Event Notification

rvf_notify class

- Generic event notification class based on *integer* identifier
- No need to declare new events
  - Simply configure them procedurally
  - No extensions
- Store event identifier in properties
  - Turn them into symbolic values

```
class rvf_xactor {
   rvf_notify notify;
   static integer BUILT;
   static integer STARTED;
   static integer STOPPED;
}
```

Event Notification

rvf_notify class

- Transactor should define events for all relevant occurrences
  - At least equivalent to callback methods
  - Before any message
  - Unlike callback, will not allow modifications
- Events must be configured
  - Broadcasted to all threads waiting (or not)
  - Persistent (or not)
  - Cannot trigger or wait on unconfigured event

```
class eth_mii extends rvf_xactor {
   static integer PRE_FR_TX = 1000;

   task new(...) {
      ...;
      void = this.notify.configure(this.PRE_FR_TX, *, *);
   }
}
```

Unique value

Broadcasted

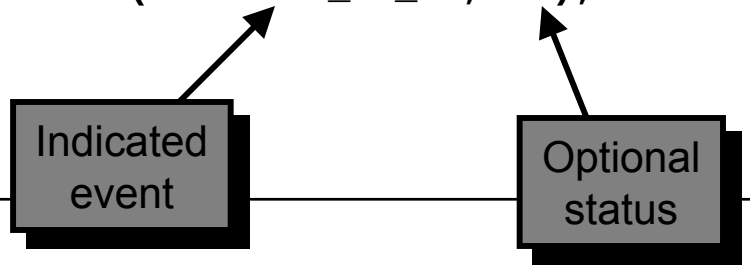
Not persistent

Event Notification

rvf_notify class

- Indicate event occurrence
  - Optionally supply "status" data structure
    - Must be *rvf_data* object
    - Can be recovered by threads waiting on notification

```
class eth_mii extends rvf_xactor {
   ...
      while (1) {
         eth_mac_frame fr;
         cast_assign(fr, this.next_transaction_t(this.to_mii));
         if (!this.pre_fr_tx_t(fr)) continue;
         this.notify.indicate(this.PRE_FR_TX, fr);
         ...
      }
   ...
}
```



Event Notification

rvf_notify class

- Any thread can synchronize with pre-defined event occurrence

```
program test {
   mii_env env = new;
   fork
      while (1) {
         void = env.mac[3].notify.wait_for_t(env.mac[3].PRE_FR_TX);
         ...
      }
   join none
   env.run_t();
}
```

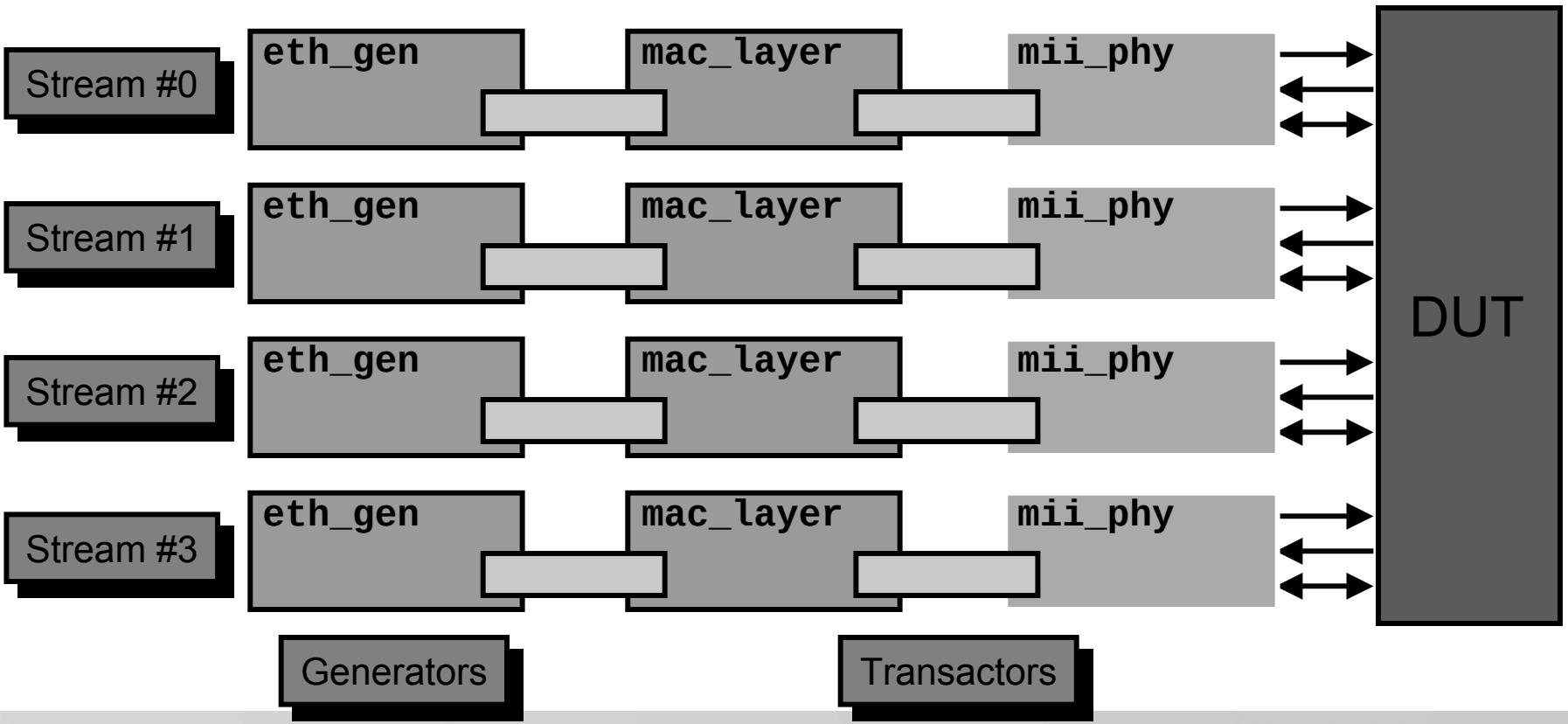
Ignore status

Indicated event

Randomizable Aspects

Data Streams

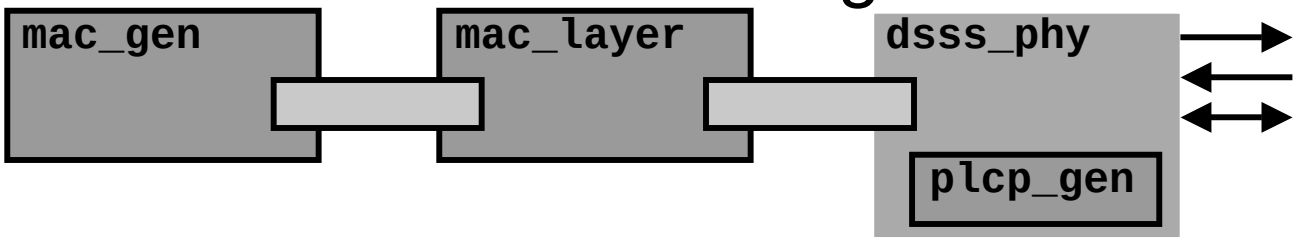
- Streams of transactions applied to DUT
- One generator per stream



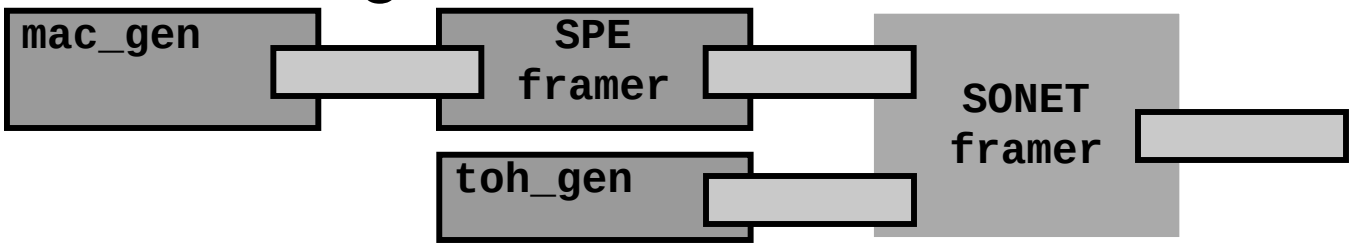
Randomizable Aspects

Incremental Data

- Additional layer-specific data added by transactor
  - Example: PLCP header in 802.11 PHY
  - Example: TOH columns in SONET/SDH
- Transactor also acts like a generator...



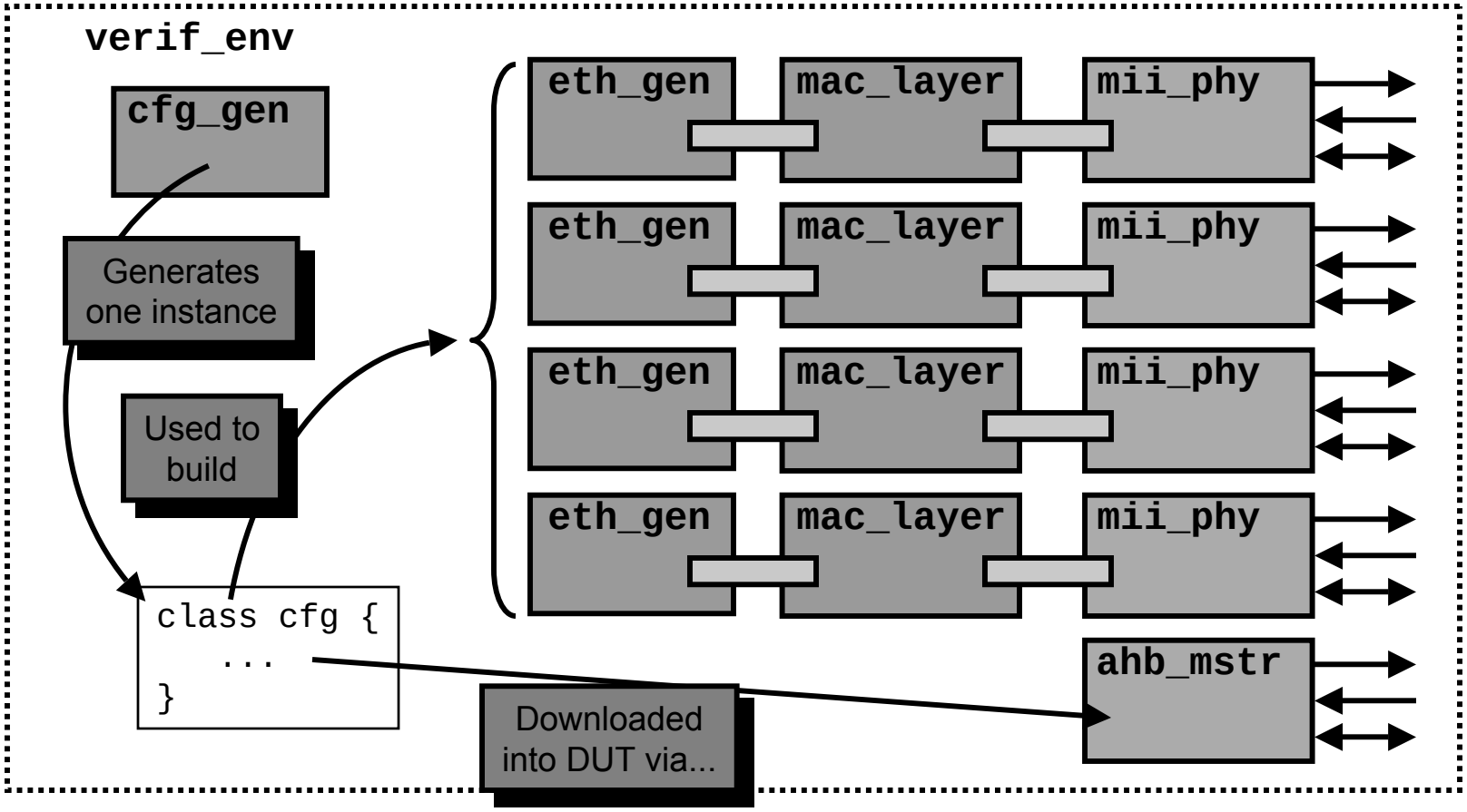
- ... Or convergence for two streams



Randomizable Aspects

Configurations

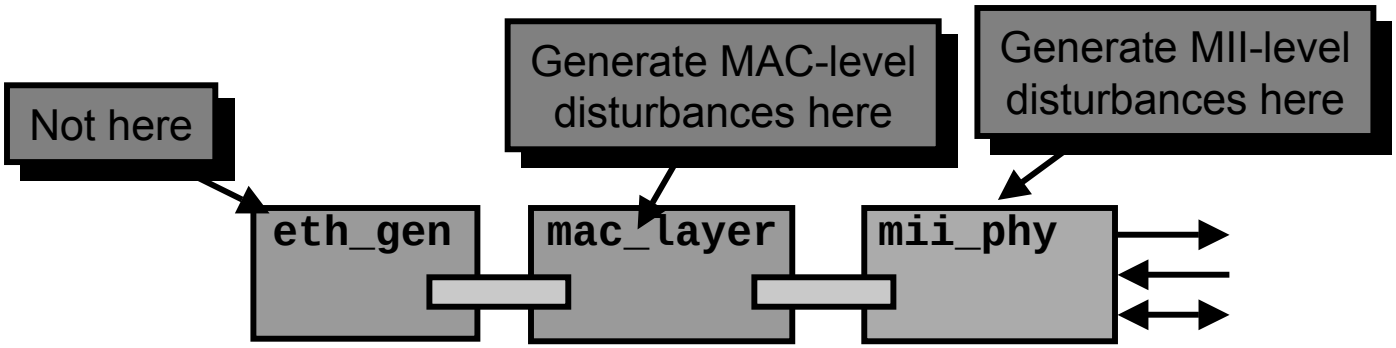
- Generated by verification environment



Randomizable Aspects

Disturbance Injection

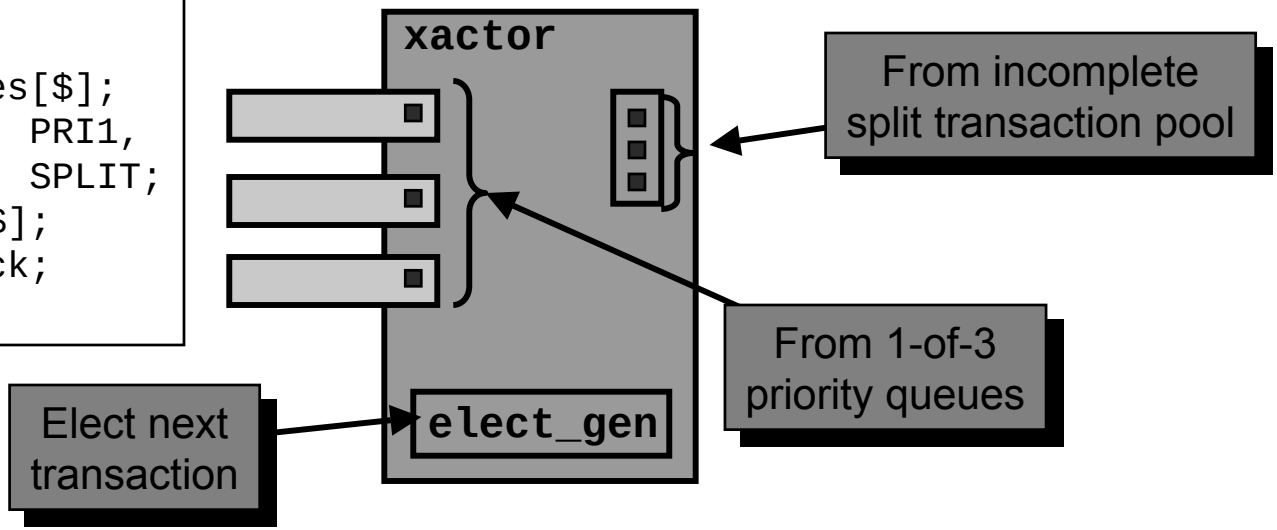
- Variations specific to a level of abstraction
  - Delay
  - Collisions
  - Abort/Retry
  - No handshake
  - Data corruption
  - Ordering
- Should be controlled in relevant transactor
  - Not at transaction source



Randomizable Aspects
Election

- Generate election result
- Randomize election descriptor
  - *Rand* property for election result
  - State properties for election candidates and controls
  - Constraints for election rules

```
class elect_transaction {
   rand integer result;

   transaction candidates[$];
   enum source_t = PRI0, PRI1,
                   PRI2, SPLIT;
   source_t     source[$];
   source_t     last_pick;
}
```



Atomic Generators

Modifying Constraints

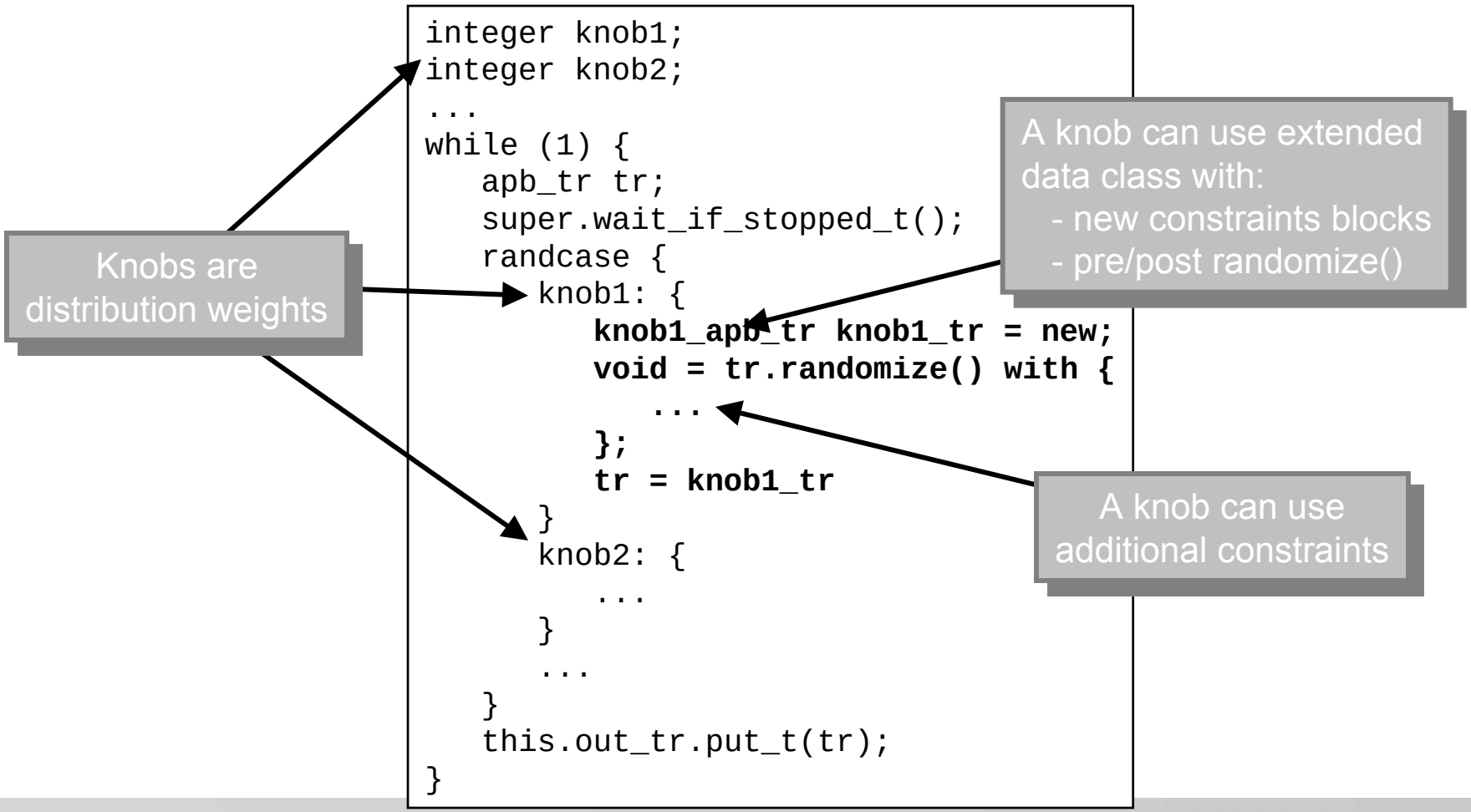
- How to modify constraints in this generator?

```
while (1) {
   apb_tr tr = new;
   super.wait_if_stopped_t();
   void = tr.randomize();
   this.out_tr.put_t(tr);
}
```

- Modify constraints in *ahb_tr* class
  - Bad: modifying generic, reusable class
  - Bad: constraints will apply to all instances
- Modify generator itself to use *randomize with*
  - Bad: modifying generic, reusable generator
  - Bad: constraints will apply to all streams
- Create new generator for each testcase
  - Bad: lots of duplicated code to maintain

Atomic Generators
Modifying Constraints

- Solution #1: knobs



Atomic Generators

Modifying Constraints

- Solution #1: knobs
  - Good: Unmodified generic, reusable transaction class
  - Good: Different generators can have different knob settings
  - Good: Knob setting can vary dynamically

  - Bad: Knob proliferation (one per corner case)
  - Bad: Generator grows with each knob
  - Bad: Knobs often turn into ON/OFF switches

Atomic Generators
Modifying Constraints

- Solution #2: polymorphism

```
apb_tr randomized_tr;
...
   while (1) {
      apb_tr tr;
      super.wait_if_stopped_t();
      void = this.randomized_tr.randomize();
      tr = this.randomized_tr.copy();
      this.out_tr.put_t(tr);
   }
```

Randomize a public static instance

Copy randomized value onto output channel

```
class my_apb_tr extends apb_tr {
   ...
}
program test {
   verif_env = new();
   env.build();
   {
      my_apb_tr my_tr = new;
      env.gen[2].randomized_tr = my_tr;
   }
   env.run_t();
}
```

Test-specific constraints in derived class

Replace randomized instance with instance of derived class

Atomic Generators
Modifying Constraints

- Solution #2: polymorphism
  - Good: Unmodified generic, reusable data class
  - Good: Unmodified generic, reusable generator class
  - Good: Different generators can have different constraint sets
  - Good: Constraint set can be dynamically changed
  - Good: Localized test-specific code
  - Good: Can use *constraint_mode()* to control constraints
  - Bad: Difficult to share constraint sets between testcases

Generators
Callback Methods

- Provide callback method
  - After new generated instance
  - Before instance is added to output channel

```
class apb_gen extends rvf_xactor {
   ...
   virtual function bit post_tr_gen_t(apb_tr tr) {
       post_tr_gen_t = 1;
   }
   ...
   virtual task main_t() {
      ...
      while (1) {
         ...
         void = this.randomized_tr.randomize();
         tr = this.randomized_tr.copy();
         if (this.post_tr_gen_t(tr)) {
            this.out_tr.put_t(tr);
         }
      }
   }
}
```

Atomic Generators
Inserting Directed Stimulus

- Directed stimulus can be added to a generator's output channel
  - Generator should be stopped

```
...
env.gen[3].stop();
{
   apb_tr tr = new;
   tr.kind = apb_tr::READ;
   ...
   env.gen[3].out_tr.put_t(tr);
}
env.gen[3].start();
```

Scenario Generators

Grammar-Based Scenarios

- Use Stream Generator

```
local task generated() {
   eth_mac_frame fr = this.randomized_fr.copy();
   if (!this.post_fr_gen_t(fr)) return;
   this.to_mac.put_t(fr);
   this.wait_if_stopped_t();
}

virtual task eth_gen::main_t() {
   ...
   while (1) {
      randseq (SCENARIOS) {

         SCENARIOS: ...
         ...
         A_FRAME: {
            void = this.randomized_fr.randomize() with {...};
            this.generated();
         }
      }
   }
}
```

Shared copy-and-forward code

Randomize with additional constraints

Could randomize from a set of instances

Scenario Generators

Grammar-Based Scenarios

- Good
  - Powerful syntax
  - Mixing of difference scenarios
  - Hierarchical scenarios
- Bad
  - Can only be controlled via knobs
  - New scenarios require modifying generator itself
    - Knobs used to select scenarios for specific test
  - Objects cannot depend on subsequent objects in scenario
  - Difficult to add functional coverage
- Use when generating simple hierarchical scenarios

Scenario Generators

Atomic Scenarios

- Constraints may need to refer to subsequent objects in scenario
- Scenario may need to be solved at once to guarantee solution will be found
- Randomize entire scenario at once instead of one element at a time

Scenario Generators
Atomic Scenarios

- Randomize scenario descriptor
  - Scenario kind & length
  - Scenario itself
    - May be hierarchical
  - Method to get next item in scenario

```
class eth_fr_scn {
   eth_mac_frame randomized_fr;

   integer             stream_id;
   integer             scenario_id;
   rand bit [31:0]     kind;
   rand bit [31:0]     length;
   rand eth_mac_frame items[$];

   virtual function eth_mac_frame next_item();
}
```

Scenario Generators
Atomic Scenarios

- Scenarios defined declaratively using constraints
- Constraint to select "valid" scenarios
- Default scenario: Single random object

```
class eth_fr_scn {
   ...
   constraint basic {
      solve kind before length;
      items.size() == length;
      if (kind == 0) length == 1;
   }

   constraint valid_scenarios {
      kind < 1;
   }
}
```

For better scenario distributions

Scenario selection constraint block

Scenario Generators
Atomic Scenarios

- Important: pre-allocate maximum-length scenario

```
class eth_fr_scn {
   ...
   protected task resize_items(integer n) {
      while (this.items.size() < n) {
         eth_mac_frame fr;
         cast_assign(fr, this.randomized_fr.copy());
         this.items.push_back(fr);
      }
   }

   task pre_randomize() {
      this.items.delete();
      this.resize_items(1);
   }
}
```

Allocate fresh instances

Maximum length default scenario == 1

Scenario Generators
Atomic Scenarios

- Apply each item from the scenario from the *next_item()* method

```
class eth_scn_gen extends rvf_xactor {
   eth_fr_scn    randomized_scn
   ...
   virtual task main_t() {
      ...
      while (1) {
         eth_mac_frame fr;
         ...
         void = this.randomized_scn.randomize();
         for (fr = this.randomized_scn.next_item();
              fr != null;
              fr = this.randomized_scn.next_item()) {
            to_mac.put_t(fr);
         }
      }
   }
}
```

Randomized instances (can be replaced)

Scenario Generators
Defining Atomic Scenarios

- Define atomic scenarios using conditional constraints
  - Based on scenario kind
    - Use *enum* to define symbolic values for scenario kinds
  - Based on frame position within scenario
  - Based on other frames in the scenario

```
class some_eth_fr_scn extends eth_fr_scn {
   enum kind_t = SHORT_BURST = 1, LONG_BURST;
   ...
   constraint valid_scenarios {
      kind <= LONG_BURST;
   }

   constraint short_burst {
      if (kind == SHORT_BURST) {
         length in {5:32};
         foreach (items, i) {
            items[i].data.size() == 42;
         }
      }
   }

   task pre_randomize() {
      super.pre_randomize();
      this.resize_items(32);
   }
   ...
}
```

Maximum scenario length

Scenario Generators
Defining Atomic Scenarios

- Redefine atomic scenarios or new scenarios in class extensions
  - Ensure scenario kind uniqueness

```
class my_eth_fr_scn extends some_eth_fr_scn {
   enum my_kind_t = CORNER = 1000;

   constraint valid_scenarios {
      kind in {LONG_BURST; CORNER};
   }

   constraint long_burst {
      if (kind == LONG_BURST) {
         length == 2;
      }
   }

   constraint corner {
      if (kind == CORNER) {
         ...
      }
   }
   ...
}
```

Redefine a scenario

Define a new scenario

Scenario Generators
Defining Directed Scenarios

- Procedural definition in *post_randomize()*

```
class my_eth_fr_scn extends eth_fr_scn {
   enum my_kind_t = DIRECTED = 1000;

   constraint valid_scenarios {
      kind in {LONG_BURST; DIRECTED};
      if (kind == DIRECTED) length == 4;
   }

   task post_randomize() {
      if (kind == DIRECTED) {
         void = rand_mode(OFF, "items");
         items[0].da = 48'h000000_123456;
         ...
         out_chan.put_t(items[0]);
         ...
         items[3].fcs = 32'h00000001;
         out_chan.put_t(items[3]);
      }
   }
   ...
}
```

Scenario Generators

Atomic Scenarios

- Good
  - Easy to redefine scenarios
  - Easy to define new scenarios
  - Can architect layers of scenarios to be reused across different test families
  - Constraints can refer to previous and subsequent objects in scenario
  - Can define hierarchical scenarios
- Bad
  - Must pre-allocate for maximum-length scenario
  - Cannot define recursive scenarios
- Use when maximum flexibility and extensibility is required

Reference Verification Flow
Summary

- Constrained-random verification methodology
- Methodology guidelines
- Base classes
- Code templates
- Training material
- Applicable to
  - Vera
  - SystemVerilog 3.1

Verification

SystemVerilog Support for Verification Methodology

Verification Technology Group
Synopsys Inc

Agenda

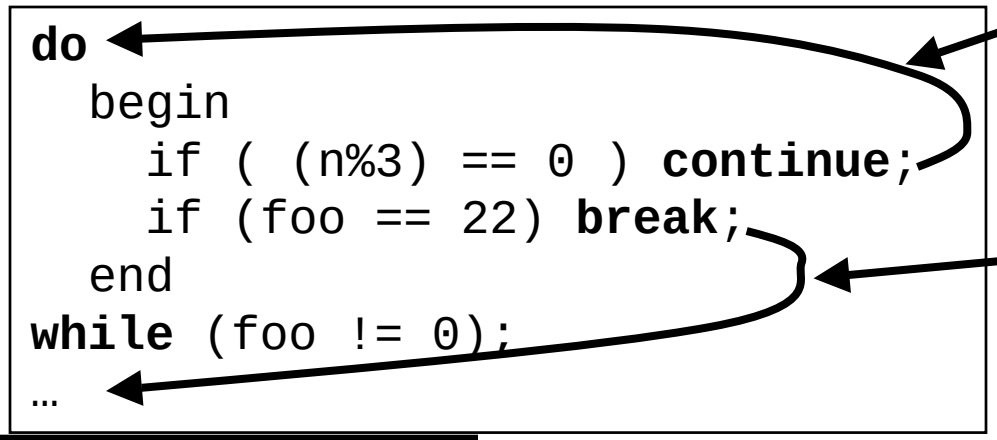
- General Modeling Enhancements
  - Datatype Extensions (structures, enums, etc)
  - Interconnect Encapsulation with Interfaces
  - Pass-by-reference
  - Specialized **always** blocks
- Testbench Infrastructure Support
  - Classes (including randomization and constraints)
  - Dynamic Memory Allocation for Arrays
  - Multiple Dynamic Processes
  - Inter-Process Synchronization and Communication
- Verification-Specific Enhancements
  - Synchronous Clocking Domains
  - Program Block
- Assertions

Basic SystemVerilog Data Types

```
reg r;      // 4-state Verilog-2001 single-bit datatype
integer i; // 4-state Verilog-2001 >= 32-bit datatype
bit b;      // single bit 0 or 1
logic w;    // 4-valued logic, x 0 1 or z as in Verilog
byte b8;    // 8 bit signed integer
int i;      // 2-state, 32-bit signed integer
shortint s;// 2-state, 16-bit signed integer
longint l; // 2-state, 64-bit signed integer
```

Explicit 2-state Variables Allow Compiler Optimizations to Improve Performance

Familiar C Features In SystemVerilog



continue starts next loop iteration

works with:
for
while
forever
repeat
do while

break exits the loop

Blocking Assignments as expressions

```
if ((a=b))…
while ((a = b || c))
```

Extra parentheses required to distinguish from **if(a==b)**

Auto increment/ decrement operators

```
x++;
if (--c > 17) c=0;
```

```
a += 3;
s &= mask;
f <<= 3;
```

Assignment Operators Semantically equivalent to blocking assignment

Wildcard Comparisons X and Z values act as wildcards

```
a =?= b
a !?= b
```

Casts And Typedefs

```
int i;
real f = 3.1515;

...

i = int'(f * 0.5);
```

Type Cast

User-defined Type

```
typedef logic [31:0] address_bus_type;

address_bus_type address_bus;
...
address_bus = address_bus_type'i;
```

Cast i as type address_bus_type

User-defined types and explicit casting improve readability and modularity

Structures

```
struct { bit [7:0]     opcode;
         bit [23:0]    addr;
       } IR;           // anonymous structure
```

IR is a struct variable

```
typedef struct { bit [7:0]    opcode;
                 bit [23:0]   addr;
} instruction;         // named structure type

instruction IR;        // define variable

IR.opcode = 1;         // set field in IR
```

instruction is a user-defined struct type

Structure definitions are just like in C but without the optional structure tag before the '{'

Packed Structures and Unions

SNUG BOSTON 2003

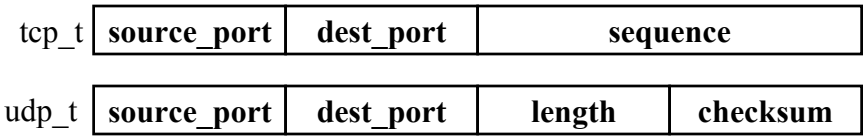
```
typedef struct packed {
  logic [15:0] source_port;
  logic [15:0] dest_port;
  logic [31:0] sequence;
} tcp_t;
typedef struct packed {
  logic [15:0] source_port;
  logic [15:0] dest_port;
  logic [15:0] length;
  logic [15:0] checksum
} udp_t;
```

```
typedef union packed {
  tcp_t tcp_h;
  udp_t udp_h;
  bit [63:0] bits;
  bit [7:0][7:0] bytes;
} ip_t;
```

All members must be the same size

```
ip_t ip_h;

ip_h.udp_h.length = 5;
ip_h.bits[31:16] = 5;
ip_h.bytes[3:2] = 5;
```

Equivalent



Create multiple layouts for accessing data

Data Organization - Enum

```
parameter IDLE = 3'b000;
parameter INIT = 3'b001;
…
typedef enum logic [2:0] {idle,
init, decode …} fsmstate;

fsmstate pstate, nstate;

case (pstate)
  idle: if (sync)
          nstate = init;
  init: if (rdy)
          nstate = decode;
…
endcase

typedef enum {lo,hi} byteloc;
memory[addr][hi] = data[hi];
```

Optional type

- Finite State Machines
  - Currently a List of Parameters
  - Why Not A Real List of Values?
  - Enumerate formally defines symbolic set of values
- Enums are strongly typed to ensure assignment to value in set
- Symbolic Indexes to Make Array References More Readable

Syntax – Implicit Named Port Connections

- Creating netlists by hand is tedious
- Generated netlists are unreadable
  - Many signals in instantiations
  - Instantiations cumbersome to manage
- Implicit port connections dramatically improve readability
- Use same signal names up and down hierarchy where possible
- Port Renaming Accentuated
- .name allows explicit connections with less typing (and less chance for error)

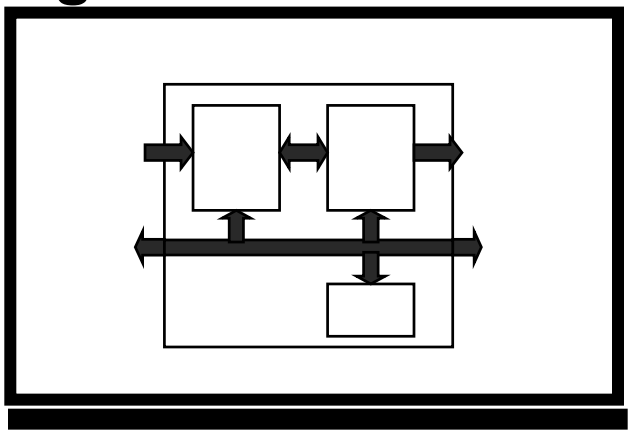
```
module top();
  logic rd,wr;
  tri [31:0] dbus,abus;
  tb(.*);
  dut(.*);
endmodule
```

```
module top();
  logic rd,wr;
  tri [31:0] dbus,abus;
  tb tb(.*, .ireset(start),
            .oreset(tbreset));
  dut d1(.*,.reset(tbreset[0]));
  dut d2(.rd, .wr, .dbus, .abus,
         .reset(tbreset[1]));
endmodule
```

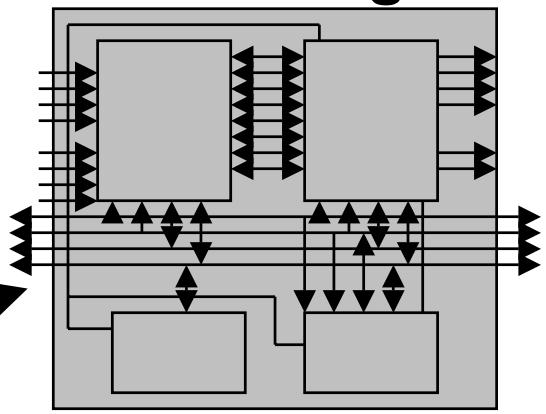
SystemVerilog Interfaces

SNUG BOSTON 2003

Design On A White Board

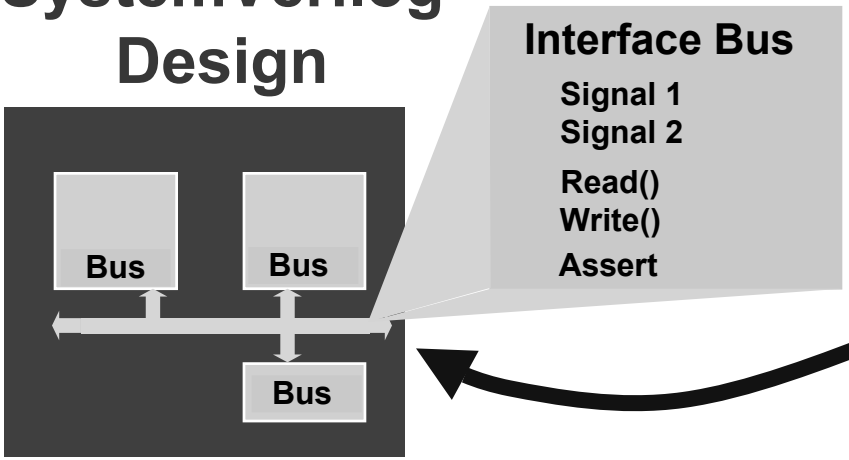


HDL Design



Complex signals
Bus protocol repeated in blocks
Hard to add signal through hierarchy

SystemVerilog Design



Interface Bus

Signal 1
Signal 2

Read()
Write()

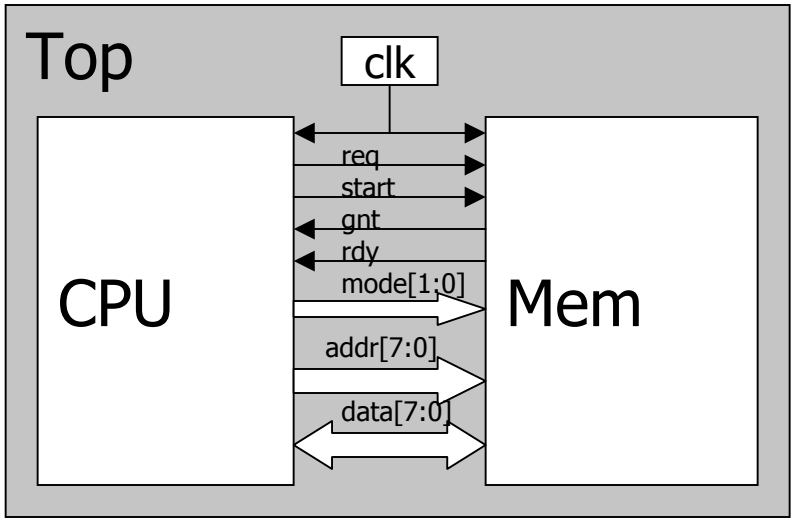
Assert

Communication encapsulated in interface
- Reduces errors, easier to modify
- Significant code reduction saves time
- Enables efficient transaction modeling
- Allows automated block verification

Example without Interface

```
module memMod(input  logic req,
                     bit clk,
                     logic start,
                     logic[1:0] mode,
                     logic[7:0] addr,
              inout  logic[7:0] data,
              output logic gnt,
                     logic rdy);
always @(posedge clk)
  gnt <= req & avail;
endmodule

module cpuMod(input  bit clk,
                     logic gnt,
                     logic rdy,
              inout  logic [7:0] data,
              output logic req,
                     logic start,
                     logic[7:0] addr,
                     logic[1:0] mode);
endmodule
```

```
module top;
  logic req,gnt,start,rdy;
  bit   clk = 0;
  logic [1:0] mode;
  logic [7:0] addr,data;

memMod mem(req,clk,start,mode,
           addr,data,gnt,rdy);
cpuMod cpu(clk,gnt,rdy,data,
           req,start,addr,mode);
endmodule
```



Example Using Interfaces

```
interface simple_bus;
  logic req,gnt;
  logic [7:0] addr,data;
  logic [1:0] mode;
  logic start,rdy;
endinterface: simple_bus

module memMod(interface a,
              input bit clk);
  logic avail;
  always @(posedge clk)
    a.gnt <= a.req & avail;
endmodule

module cpuMod(interface b,
              input bit clk);
endmodule
```

Bundle signals in interface

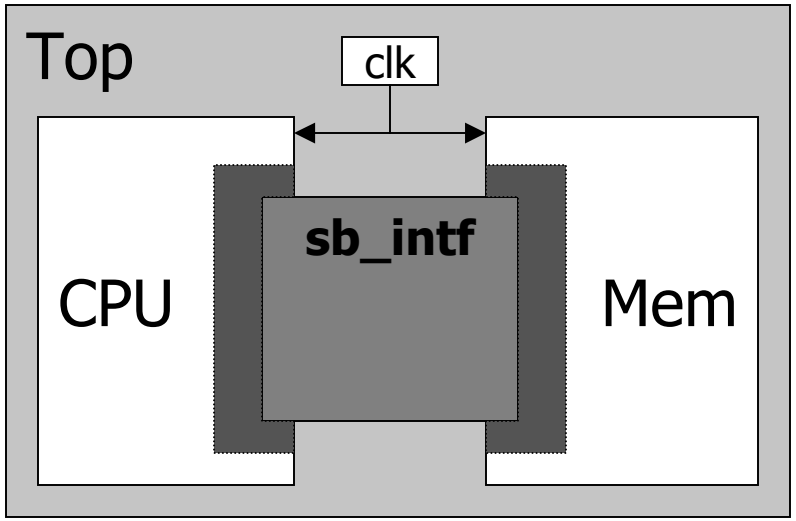
Use interface keyword in port list

Refer to intf signals

```
module top;
  bit clk = 0;
  simple_bus sb_intf;

  memMod mem(sb_intf, clk);
  cpuMod cpu(.b(sb_intf),
             .clk(clk));
endmodule
```

interface instance

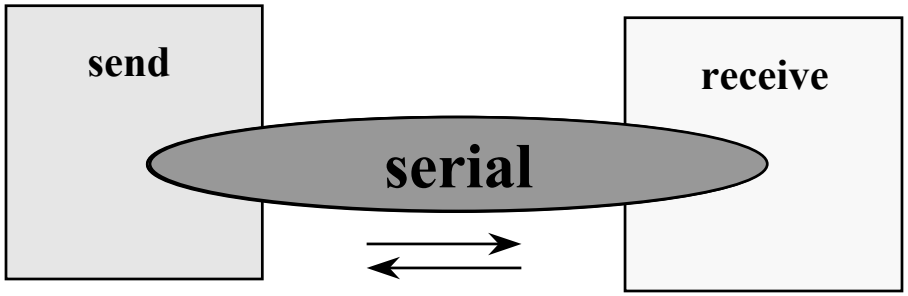
Connect interface



Using Different Interfaces

```
typedef logic [31:0]
data_type;

bit clk;
always #100 clk = !clk;

parallel channel(clk);
send    s(clk, channel);
receive r(clk, channel);
```

```
typedef logic [31:0]
data_type;

bit clk;
always #100 clk = !clk;

serial channel(clk);
send    s(clk, channel);
receive r(clk, channel);
```




```
module send(input bit clk,
            interface i);
  data_type d;
...
  i.write(d);
endmodule
```

Module inherits communication method from interface

Simplifies design exploration
Extends block-based design to the communication between blocks

Task and Function Arguments

Default Arguments

```
task foo(input int a=1,
                   b=2);
```

Usage

```
foo(); // use default values
foo(3); // foo.a=3,foo.b = default
foo(.b(4)); // foo.a = default,
                   foo.b=4
```

Connect by name, too

Pass by Reference

```
task foo(ref event c,
         const ref d);
  @(c) req <= 1;
  @(d) req <= 0;
  ...
endtask
```

Read-only argument

Sensitivity to task args

Usage

```
foo(posedge clk, my_sig);
```

Event passed as argument

Function Arguments

```
function void bar(inout int a);
  a = a+1;
endfunction
```

No return value

inout arg

Usage

```
begin
  b = 2;
  bar(b); // b = 3
end
```

function as 0-time statement

Design Intent – always_{comb,latch,ff}

- always blocks do not guarantee capture of intent
- If not edge-sensitive then only a warning if latch inferred

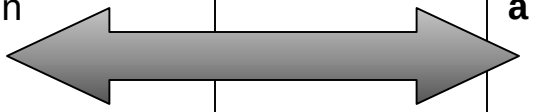
- always_comb, always_latch and always_ff are explicit
- Compiler Now Knows User Intent and can flag errors accordingly

```
//OOPS forgot Else but it's
//only a synthesis warning
always @(a or b)
  if (b) c = a;



//Compiler now asks
//"Where's the else?"
always_comb
  if (b) c = a;
//Intent: Conditional
//        Assignment
always_latch
  if (clk)
   if (en) Q <= d;
//Conversely unconditionally
//assigned –is it a latch?
always_latch
     q <=d
```

Design Intent – `always_comb` Sensitivity

- always_comb eliminates sensitivity list issues
  - Ensures synthesis-compatible sensitivity
  - Helps reduce “spaghetti code”
- Consider that always_comb derives sensitivity from
  - RHS/expr in process
  - RHS/expr of statements in Function Calls

```
logic avar,a,b,c,d,e;
logic [1:0] sel;
always_comb begin
    a = b;
    StepA();
end
function StepA
  case (sel)
    2'b01: avar = a | c;
    2'b10: avar = d & e;
    default: avar = c;
  endcase
endfunction
```



```
always @(sel,b,c,d,e) begin
  a = b;
  case (sel)
    …
  endcase
end
```

Encapsulate blocks of combinational logic into functions – Easier to read and debug

Final Block

- Triggered by $finish, PLI tf_dofinish
- Same code restrictions as in functions
  - no delays, event controls, or task calls

```
final
  begin
    $display("Instructions executed %d", icount);
    if (ACC == 16'hbeef)
      $display("ACC has correct result");
    else
      $display("ACC has incorrect result");
  end
```

Useful for statistics, coverage and file closure

Verification

SystemVerilog

TestBench Infrastructure Support

Classes – SystemVerilog & Vera

SystemVerilog

```
class name;
<data_declarations>;
<task/func_decls>;
endclass
```

```
class Packet;
  rand bit[3:0] cmd;
  int status;
  myStruct header;

  constraint c1 { cmd[0] == 1'b0; }

  function int get_status();
    return(status);
  endfunction
  extern task set_cmd(input bit a);
endclass
```

```
task Packet::set_cmd(input bit a);
  cmd = a;
endtask
```

Vera

```
class name {
<data_declarations>;
<task/func_decls>;
}
```

```
class Packet {
  rand bit[3:0] cmd;
  int status;
  myClass header;

  constraint c1 { cmd[0] == 1'b0; }

  function int get_status(){
    return(status);
  }
  task set_cmd(input bit a);
}
```

```
task Packet::set_cmd(input bit a){
  cmd = a;
}
```

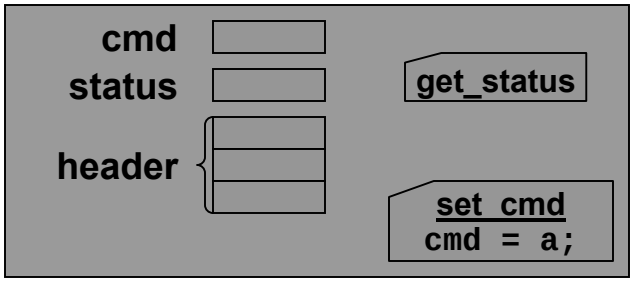
Familiar to Vera users
Natural extension for Verilog users

Class Inheritance & Extension

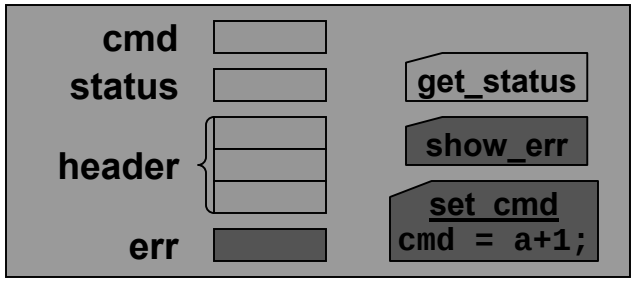
- Keyword ***extends*** Denotes Hierarchy of Definitions
  - Subclass inherits properties, constraints and methods from parent
  - Subclass can redefine methods explicitly

```
class ErrPkt extends Packet;
  bit[3:0] err;
  function bit[3:0] show_err();
    return(err);
  endfunction
  task set_cmd(input bit[3:0] a);
    cmd = a+1;
  endtask // overrides Packet::set_cmd
endclass
```

Packet:



ErrPkt:



Allows Customization Without Breaking or Rewriting Known-Good Functionality in the Base Class

Dynamic Arrays

Declaration syntax

```
<type> <identifier> [ ];
bit[3:0] dyn[ ];
```

Initialization syntax

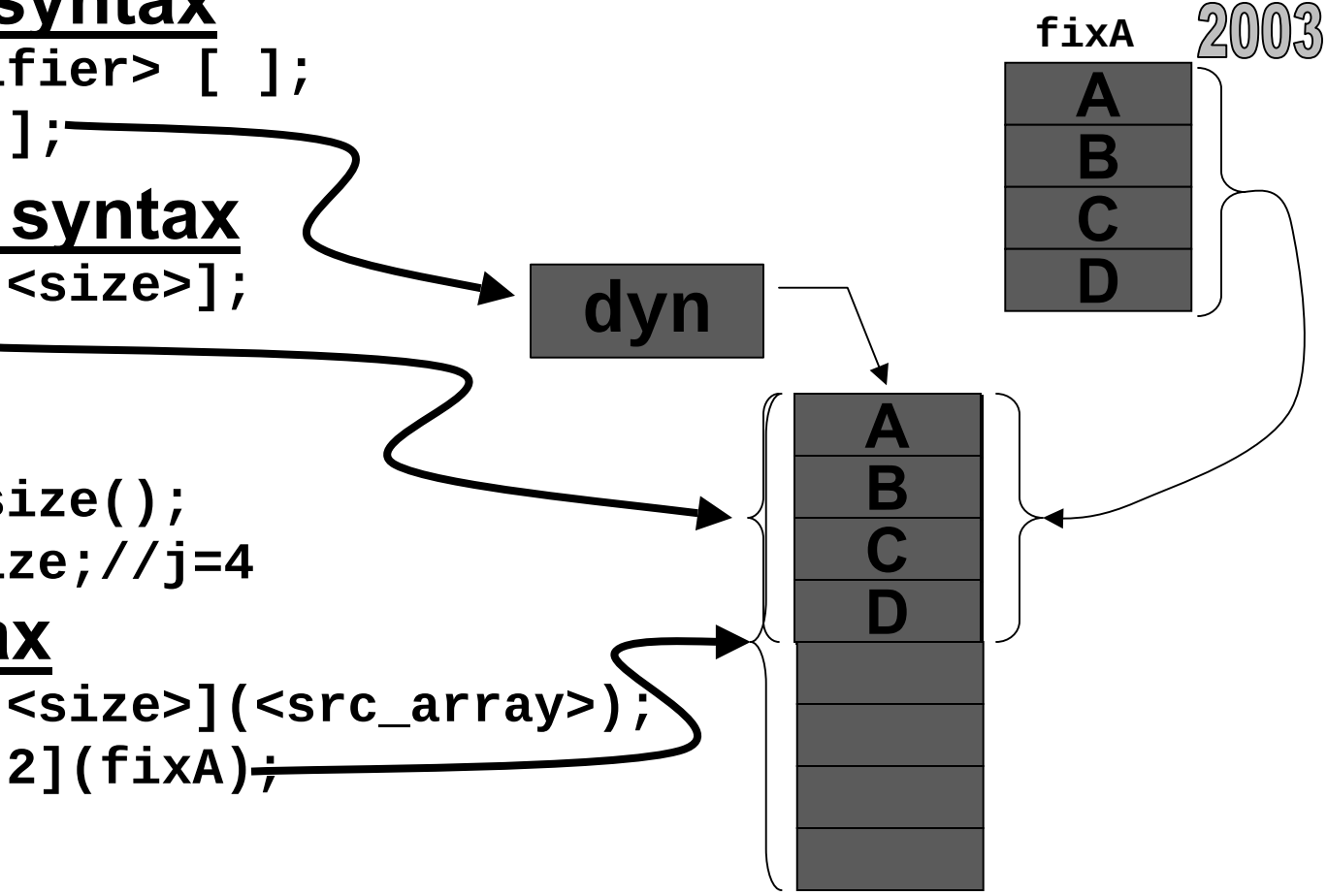
```
<array> = new[<size>];
dyn = new[4];
```

Size method

```
function int size();
int j = dyn.size;//j=4
```

Resize syntax

```
<array> = new[<size>](<src_array>);
dyn = new[j * 2](fixA);
```



Dynamic Arrays

Declaration syntax

```
<type> <identifier> [ ];
bit[3:0] dyn[ ];
```

Initialization syntax

```
<array> = new[<size>];
dyn = new[4];
```

Size method

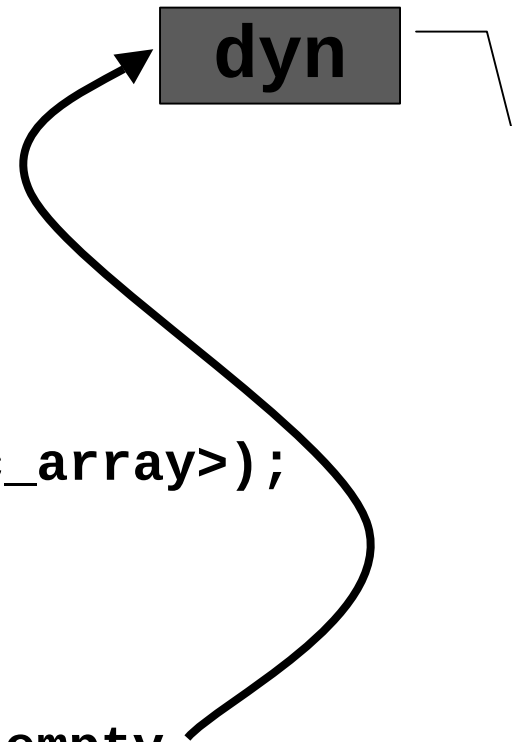
```
function int size();
int j = dyn.size;//j=4
```

Resize syntax

```
<array> = new[<size>](<src_array>);
dyn = new[j * 2](fix);
```

Delete method

```
function void delete();
dyn.delete; // dyn is now empty
```



Associative Arrays

- Sparse Storage
- Elements Not Allocated Until Used
- Index Can Be of Any Packed Type, String or Class

Declaration syntax

```
<type> <identifier> [<index_type>];
<type> <identifier> [*]; // "arbitrary" type
```

Example

```
struct packed {int a; logic[7:0] b} mystruct;
int myArr [mystruct]; //Assoc array indexed by mystruct
```

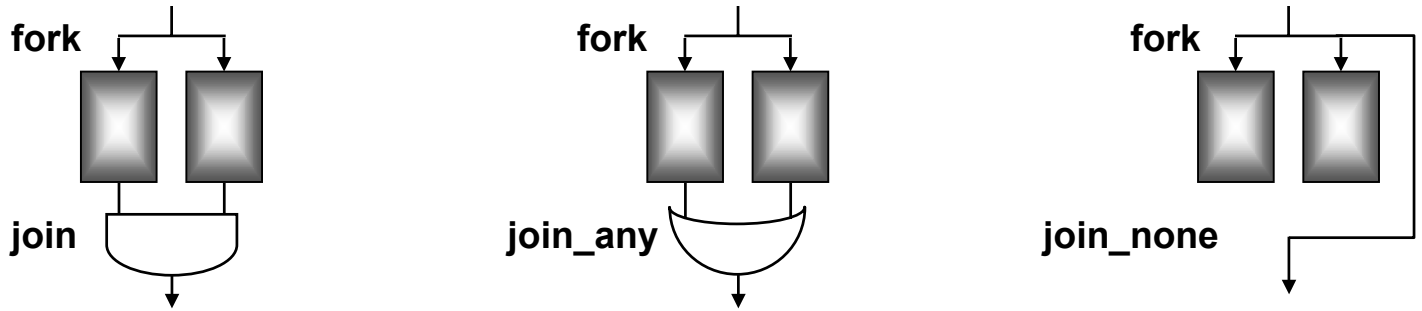
Built-in Methods

```
num(), delete([index]), exists(index);
first/last/prev/next(ref index);
```

Ideal for Dealing with Sparse Data

Dynamic Processes and Threads

- SystemVerilog adds dynamic parallel processes using **`fork/join_any`** and **`fork/join_none`**



- Threads created via fork…join
- Threads execute until a blocking statement
  - wait for: (event, mailbox, semaphore, variable, etc.)
  - disable fork to terminate child processes
  - wait_child to wait until child processes complete
- $exit terminates the main program thread

Multiple Independent Threads
Maximize Stimulus Interactions

Inter-Process Synchronization and Communication

- Events – enhanced from V2K
  - Events are variables – can be copied, passed to tasks, etc.
  - event.triggered; // persists throughout timeslice, avoids races
  - wait_order(), wait_any(), wait_all(<events>);
- Semaphore – Built-in Class
  - Built-in methods: get, put, try_get

```
int semID = alloc(SEMAPHORE,0,1,1);
semaphore_get(WAIT,semID,1);
semaphore_put(semID,1);
```

vs.

```
semaphore semID = new(1);
semID.get(1);
semID.put(1);
```

- Mailbox – Built-in Class
  - Built-in methods: num(), put(), try_put(), get(), try_get(), peek(), try_peek()
  - Arbitrary type

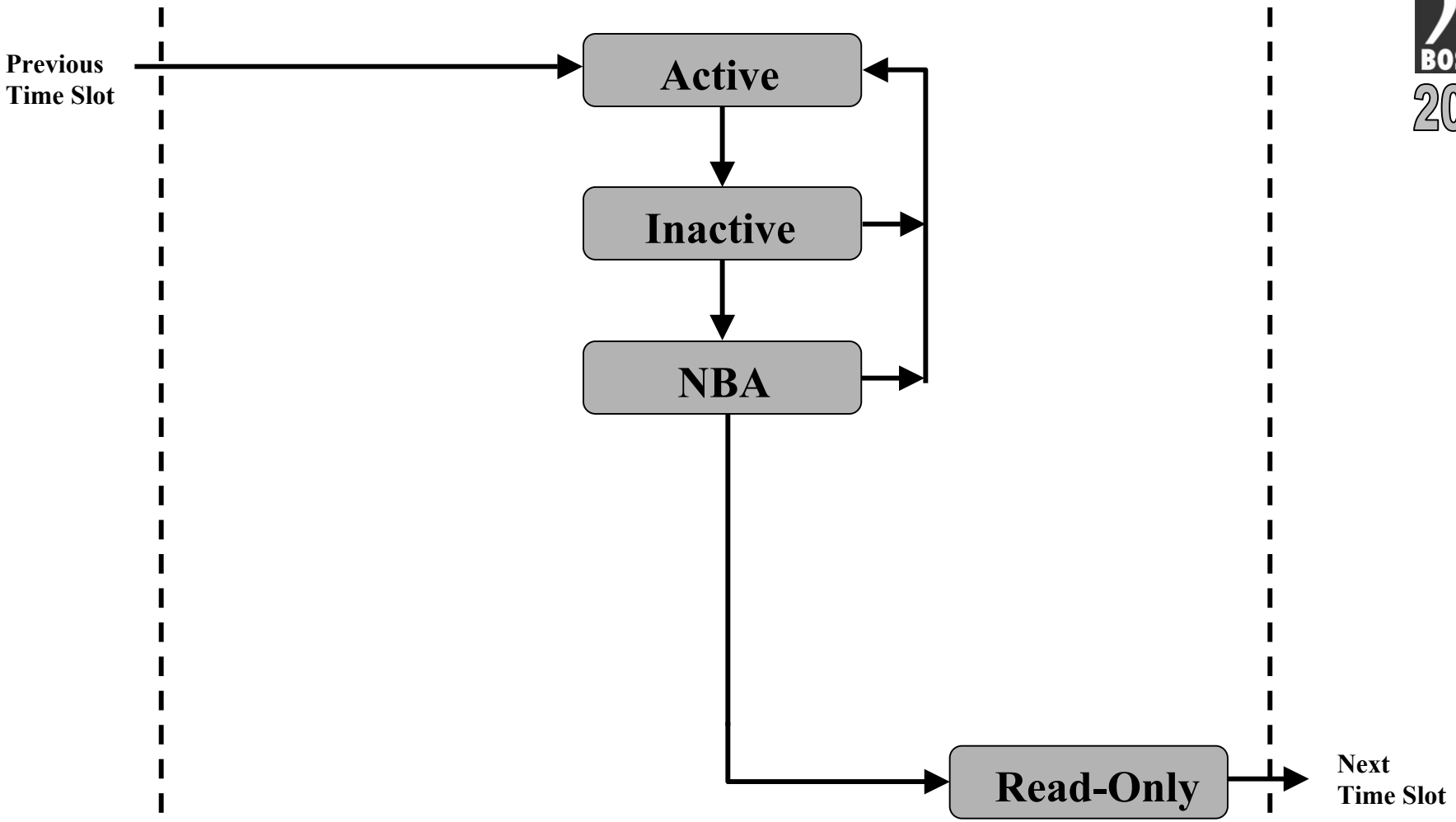
```
int mbID = alloc(MAILBOX,0,5);
cnt = mailbox_get(WAIT,mbID,msg);
mailbox_put(mbID,msg);
```

vs.

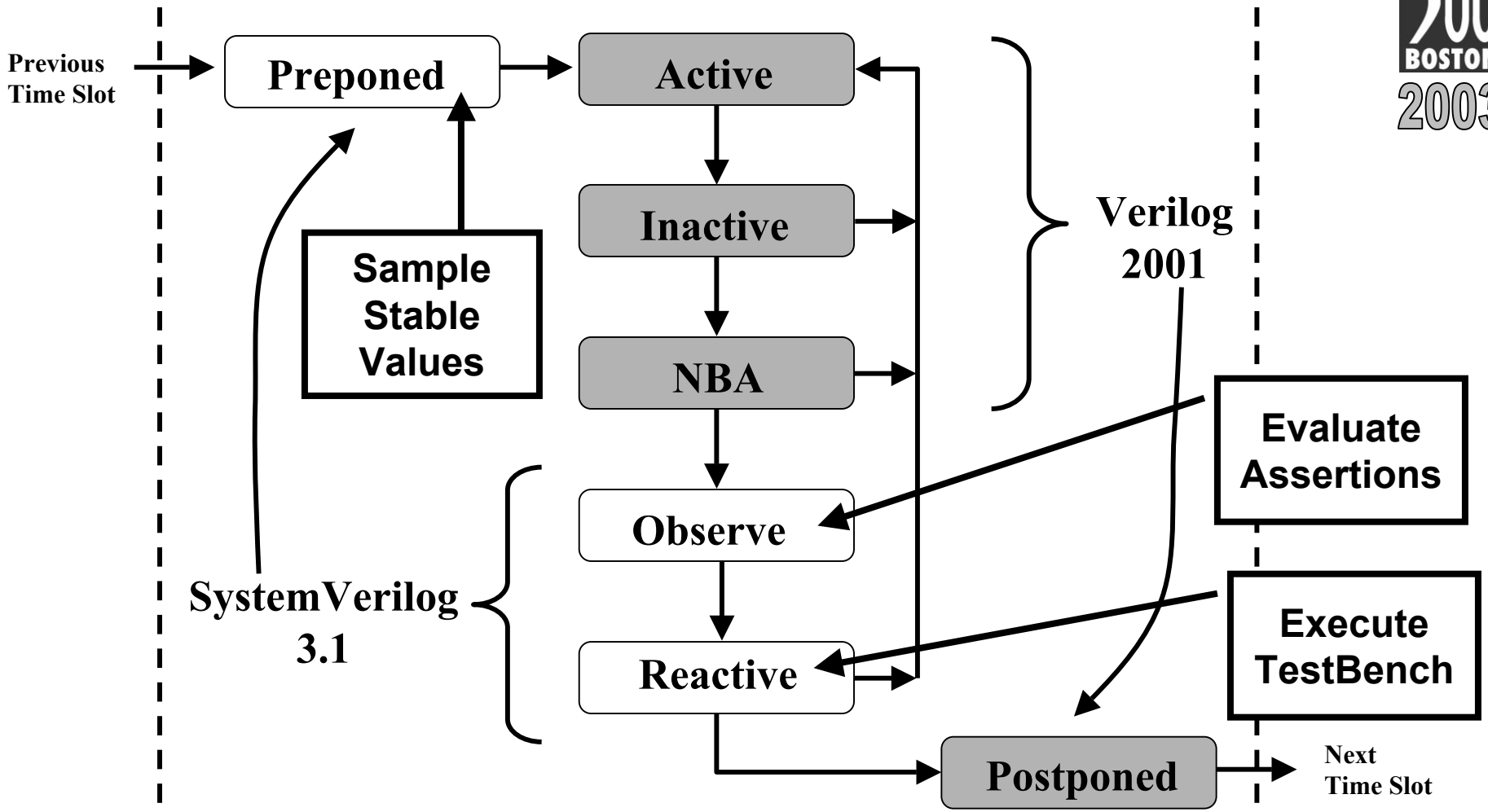
```
mailbox #(type) mbID = new(5);
mbID.get(msg);
mbID.put(msg);
```

Ensures meaningful, race-free communication between processes

SystemVerilog Enhanced Scheduling



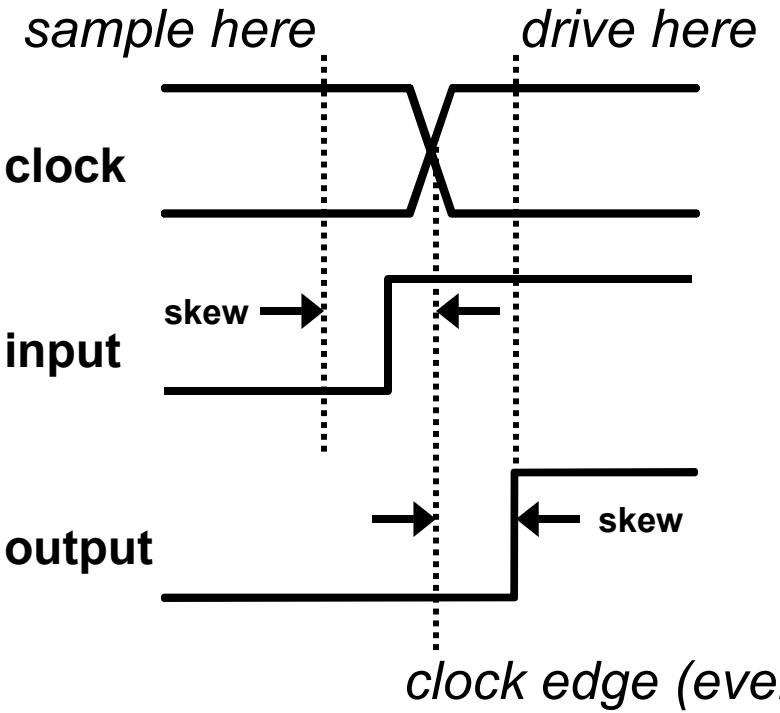
SystemVerilog Enhanced Scheduling



Enhanced Scheduling allows assertions and testbench to work together without drastically altering scheduling algorithms

Skew Declaration

- Specify Synchronous Sample and Drive Times



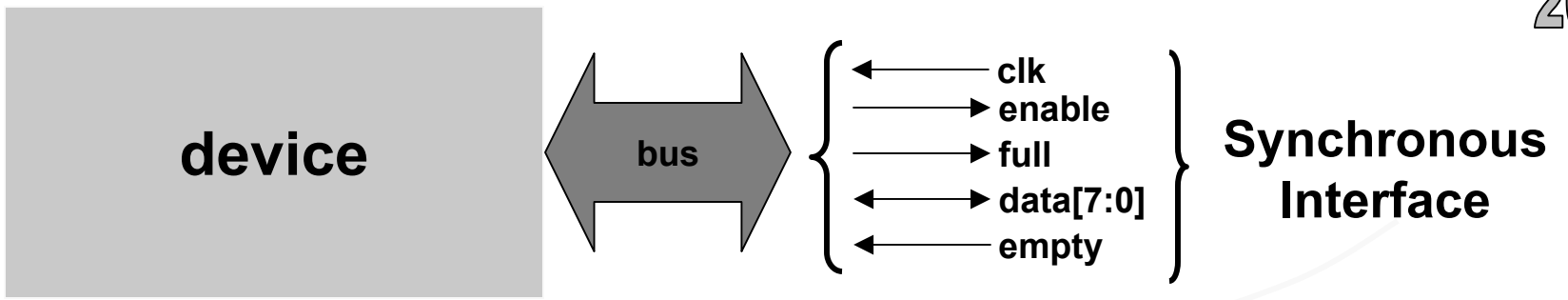
Input skew is for sampling

Output skew is for driving

*Default input skew is **1step***
Default output skew is 0.

Ensure race-free interactions between testbench and design

Synchronous Interfaces: Clocking




```
clocking bus @(posedge clk);
  default input #1ns output #2ns;

  input          enable, full;
  inout          data;
  output         empty;
  output  #6ns   reset = top.u1.reset;
endclocking
```

Clocking Event "clock"

Default I/O skew

Hierarchical signal

Override Output skew

```
Testbench Uses:
 bus.enable
 bus.data
  ...
```

Default Clocking and Synchronous Drives

- Designate one clocking as default
  - default clocking modA.clkDomain;
- One default per module, interface, program
- Cycle Delay Syntax:
  - ## <integer_expression>
  - ##5; // wait 5 cycles
  - ##1 bus.data <= 8'hz;// wait 1 (bus) cycle and then drive data
  - ##2; bus.data <= 2;// wait 2 default clocking cycles, then drive data
  - bus.data <= ##2 r;// remember the value of r and then drive data 2 (bus) cycles later

Program Block

- Purpose: Identifies verification code
- A **`program`** differs from a **`module`**
  - Only initial blocks allowed
  - Special semantics
    - Executes in *Reactive* region

      design → clocking/assertions → program

```
program name (<port_list>);
  <declarations>; // type, func, class, clocking…
  <continuous_assign>
  initial <statement_block>
endprogram
```

The Program block functions pretty much like a C program
Testbenches are more like software than hardware

SystemVerilog Testbench Language Summary

- Testbench Extensions
  - Useful for modeling environment
    - Classes
    - Random Constraints
    - Pass-by-Reference
    - Process Control
      - Interprocess communication
      - Synchronization
    - Extended data types
- Verification Extenstions
  - Enhanced Scheduling
  - Program Block
  - Clocking Domains
  - Assertions

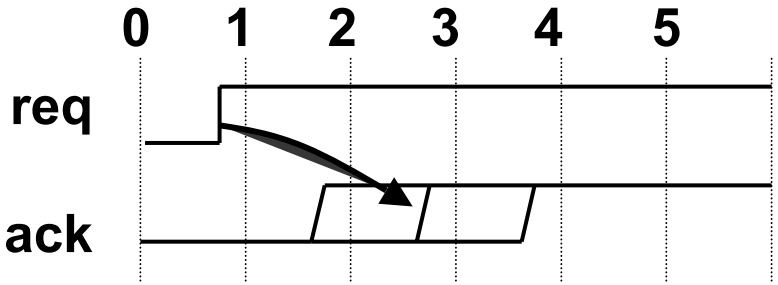
Verification

SystemVerilog Assertions

SNUG BOSTON 2003

What is an Assertion?

A concise description of [un]desired behavior



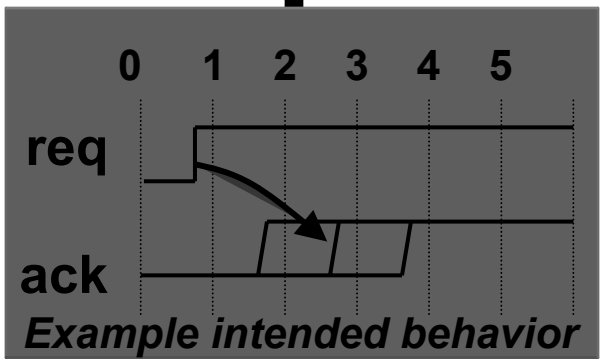
Example intended behavior

"After the request signal is asserted, the acknowledge signal must come 1 to 3 cycles later"

Concise and Expressive SVA

SVA Assertion

```
property req_ack;
 @(posedge clk) req ##[1:3] $rose(ack);
endproperty
as_req_ack: assert property (req_ack);
```

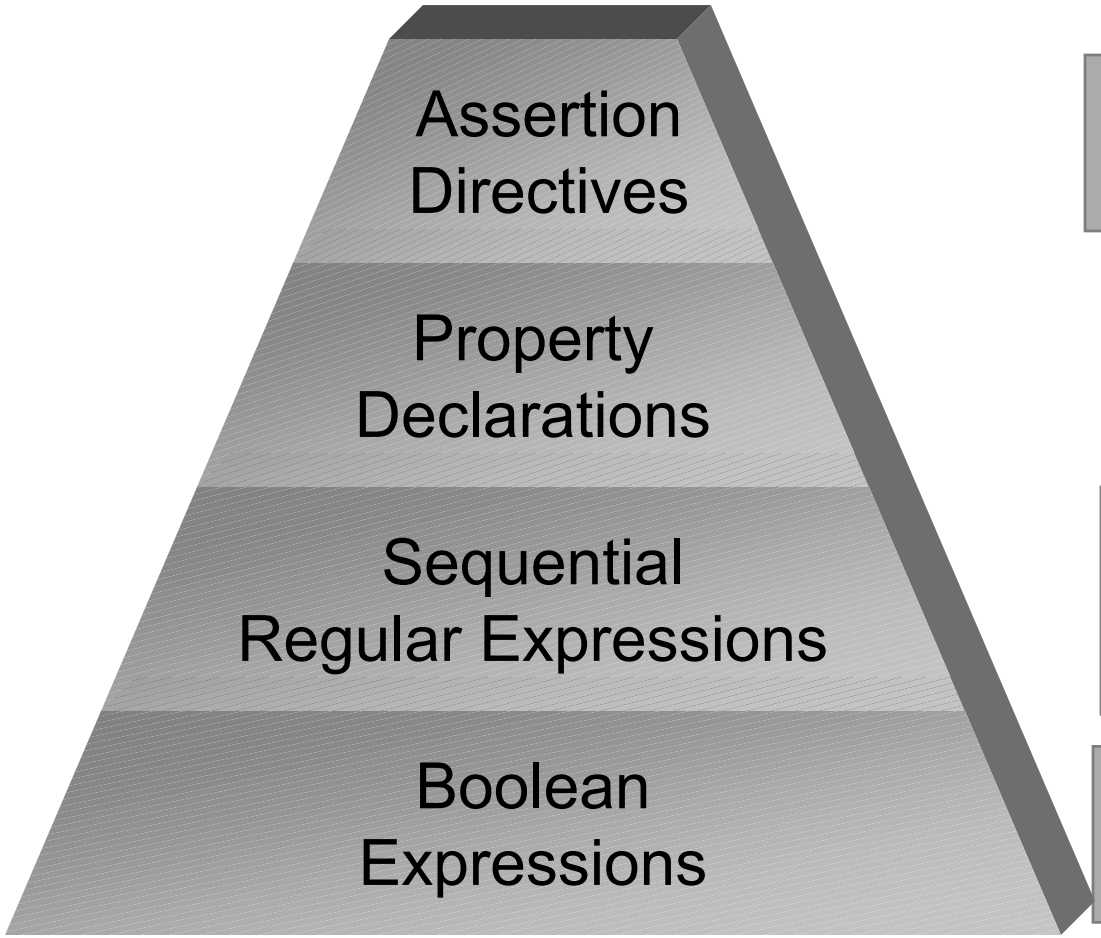


HDL Assertion

Verilog

```
always @(posedge req)
  begin
     repeat (1) @(posedge clk);
     fork: pos_pos
        begin
           @(posedge ack)
           $display("Assertion Success",$time);
            disable pos_pos;
        end
       begin
          repeat (2) @(posedge clk);
          $display("Assertion Failure",$time);
          disable pos_pos;
       end
     join
  end // always
```

Hierarchy of Assertion Constructs




```
assert, cover, bind,
declarative instantiation,
procedural instantiation
```

```
disable iff, not,
implication
```

```
repetition,(cycle)delay,
and, or, intersect,
first_match,
within, throughout
```

```
<expr>, <function>,
<temporal_edge_function>,
ended, matched
```

Each layer builds on the previous layer

Embedding Concurrent Assertions

```
sequence s1;
   (req && !gnt)[*0:5] ##1 gnt && req ##1 !req ;
endsequence

always @(posedge clk or negedge reset)
  if(reset == 0) do_reset;
  else if (mode == 1)
    case(st)
    REQ: if (!arb)
              if (foo)
                 st <= REQ2;
                 PA: assert property (s1);
```

- Automatically Updates Enabling Condition as Design Changes
- Infers clock from instantiation

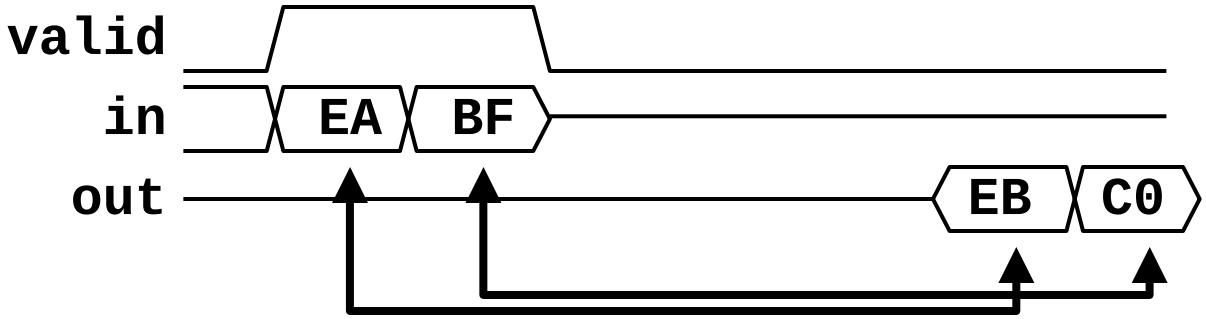
- Requires User to Update Manually as Design Changes

```
property p1;
  @(posedge clk) ((reset == 1) && (mode == 1)
                        && (st == REQ) && (!arb) && (foo)) => s1;
endproperty

DA: assert property (p1);
```

Manipulating Data: Local Dynamic Variables

- Declared Locally within Sequence/Property
  - New copy of variable for each sequence invocation
- Assigned anywhere in the sequence
- Value of assigned variable remains stable until reassigned in a sequence

Local Dynamic Variable Example



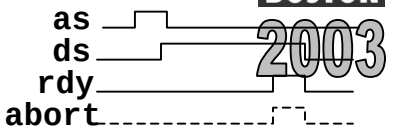

```
property e;
int x;
(valid,(x=in))|=> ##5(out==(x+1));
endproperty
```

Flexible Assertions Use-Model

- Design Engineers
  - Able to define assertions in-line with design code
  - Assertions typically cover implementation-level detail
  - Capture assumptions while they're fresh in the designer's mind
- Verification Engineers
  - Able to define assertions external to RTL code and "bind" them to the design
  - Assertions typically cover interface/system behavior
  - Do not need to modify the golden RTL to add assertions

TB + Assertions Example

A new bus cycle may not start for 2 clock cycles after an abort cycle has completed




```
sequence abort_cycle;
 !rdy throughout (as ##1 ds[*1:$] ##1 abort);
endsequence
```

```
cover property (@(posedge clk) abort_cycle)
   wait_cnt = 2;
```

```
property wait_after_abort;
  @(posedge clk) abort_cycle |=> !as[*2];
endproperty
assert property (wait_after_abort);
```

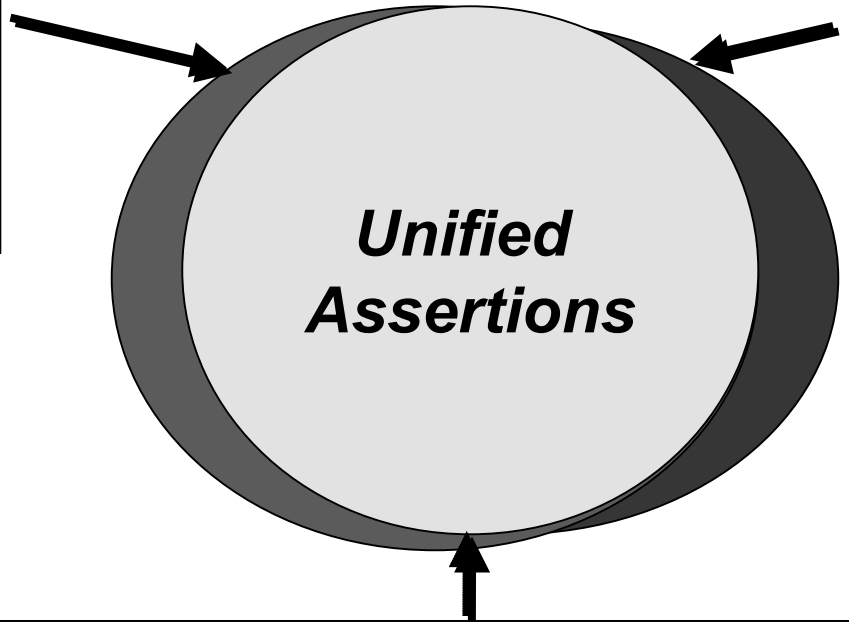
Simulation Monitors and Constraints for Formal Analysis

```
program manual_stimulus_generator;
repeat(1000) begin
  generate_transaction(addr,data);
  while(wait_cnt > 0)
    @(posedge clk) wait_cnt--;
  end
endprogram
```

Roadmap – Accellera Assertions

SVA 3.1
- *no change!*
- *Unified!*

PSL 1.1
- *Under development*



Unified Assertions
- Common subset of SVA and PSL
- Syntax and semantics based on SVA
- Compatible for formal and simulation
- Contains most assertion language features

Source: Accellera website
http://www.accellera.org/TCC-Report-DAC2003.pdf

Testbench/DUT Infrastructure

Option 1:

```
class myPkt;
...
endclass
```

Passing the intf to the program isolates the TB from abstraction changes in the DUT

Keeping the extended class definition in the interface simplifies file management and guarantees consistency

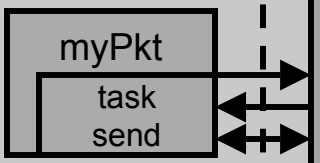
Top

```
module top;
  intf1 intf; // instantiate interface
  myProg myP(intf1); // instantiate program
  DUT myDUT(intf1);
endmodule
```

```
program myProg(intf i);
initial begin
  i.myPkt.send();
  ...
  end
endprogram
```

intf1

```
myPkt myPkt = new
```



RTL DUT

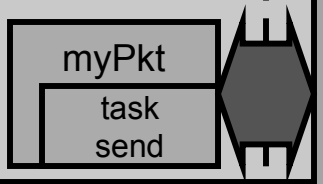
Top

```
module top;
  intf2 intf; // instantiate interface
  myProg myP(intf1); // instantiate program
  DUT myDUT(intf1);
endmodule
```

```
program myProg(intf i);
initial begin
  i.myPkt.send();
  ...
  end
endprogram
```

intf2

```
class myTLPkt extends myPkt;
...
endclass
myTLPkt myPkt = new;
```



TL DUT

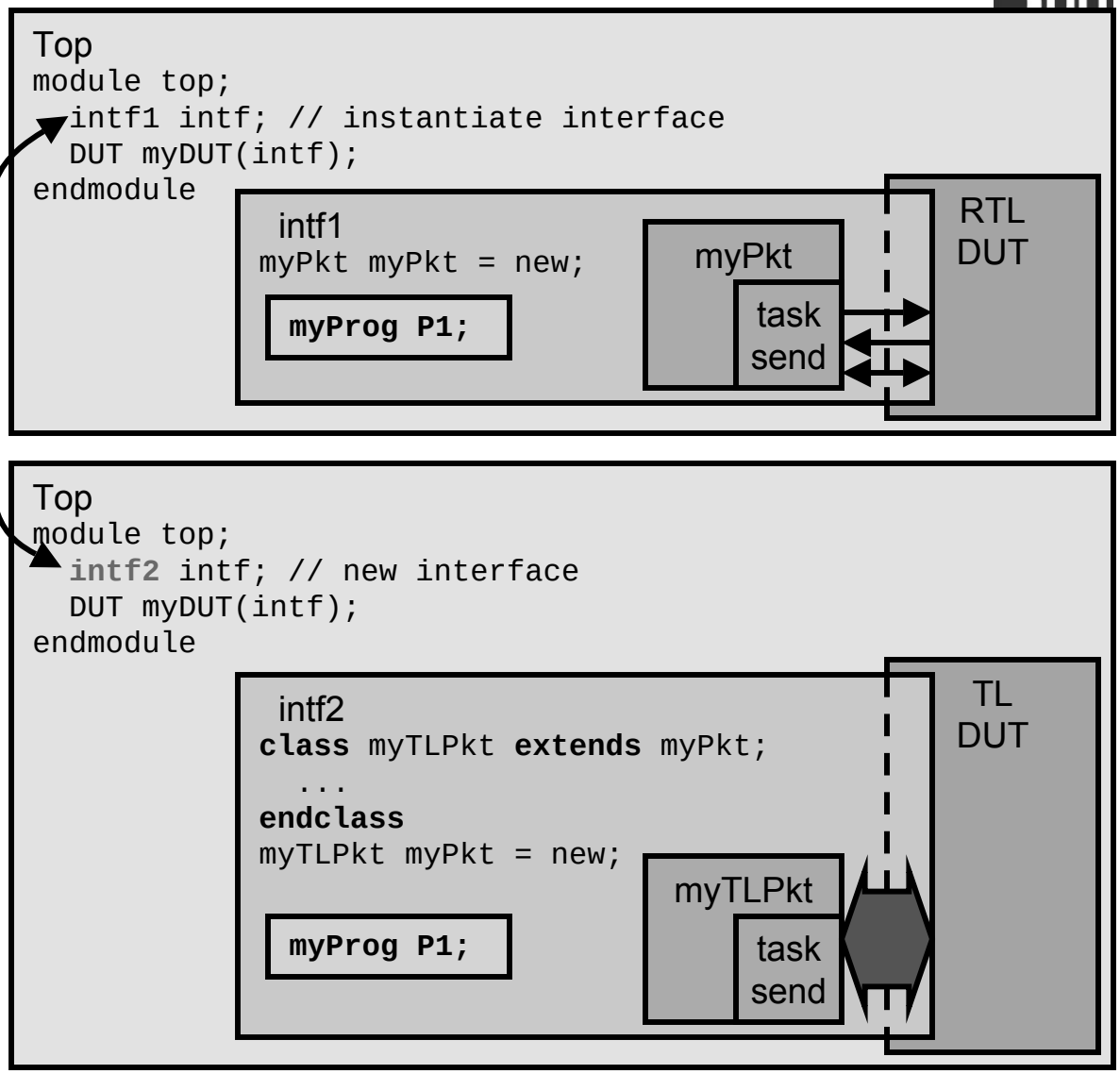
Testbench/DUT Infrastructure

Option 2:

```
class myPkt;
...
endclass
```

```
program myProg;
initial begin
  myPkt.send();
  ...
  end
endprogram
```

Instantiating the program directly in the interface makes the interface self-testing



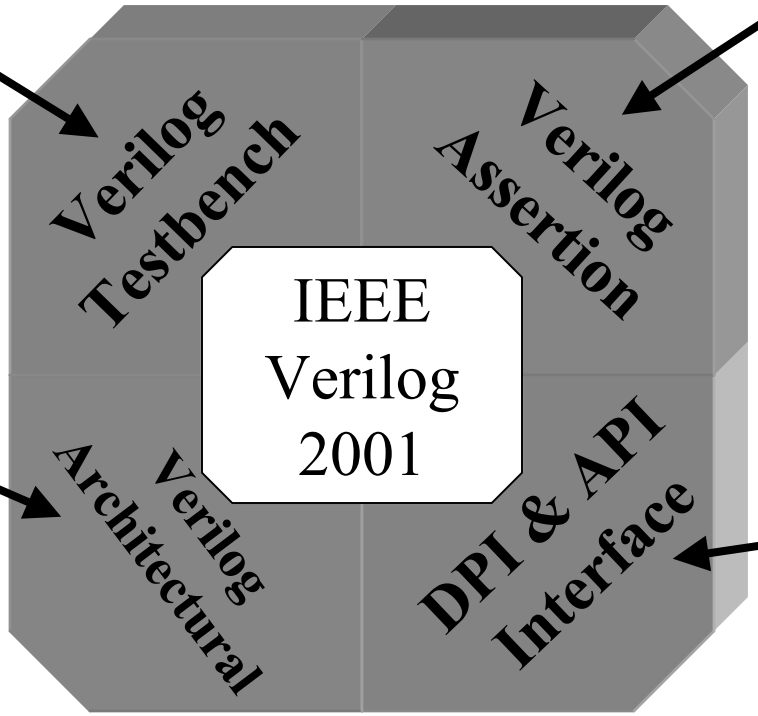
Advantages of SystemVerilog

- Vera/TB Functionality with Verilog-compatible syntax
  - Easy to build on existing Verilog testbenches
  - Supports layering/inheritance for advanced verification environments
  - No need to change existing Vera-based methodologies
- SystemVerilog Interfaces simplify design exploration
  - Change abstraction levels
  - Encapsulate assertions for self-checking protocols
- Mailboxes and Semaphores built into language
  - Mailboxes support arbitrary types
  - Method-based syntax for ease of use
- Straightforward TB/DUT connection mechanism
  - No artifical name binding layer
  - Hierarchical references supported directly

SystemVerilog Summary

- Extends Verilog IEEE 2001 to higher abstraction levels for Architectural and Algorithmic Design , and Advanced Verification.

Transaction-Level Full Testbench Language with Coverage



Advanced verification capability for semiformal and formal methods. The Assertion Language Standard For Verilog

Design Abstraction: Interface semantics, abstract data types, abstract operators and expressions

Direct C interface, Assertion API and Coverage API