Creating Stimulus and Stimulating Creativity:

Using the VMM Scenario Generator

Jonathan Bromley

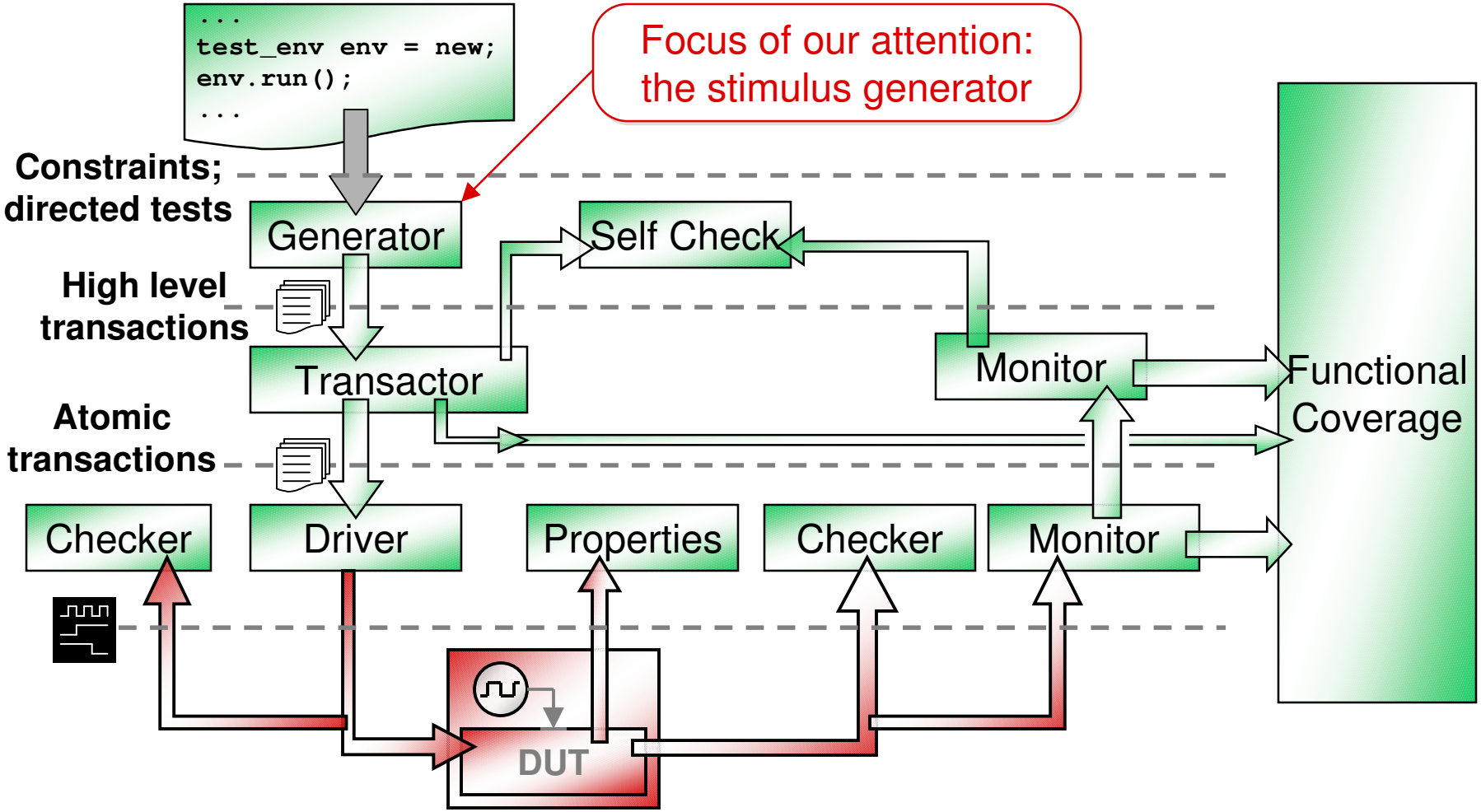
Doulos Ltd, Ringwood, UK

jonathan.bromley@doulos.com

Outline

- Introduction: motivation for scenarios
- The VMM Standard Library scenario generator
- Motivation for hierarchical scenarios
- Prototype VMM-compatible hierarchical scenarios
- Further opportunities and future work
- Summary

VMM testbench architecture



Motivation for scenarios

- VMM transactions can be randomized
  - custom constraints
- Realistic DUTs need structured sequences of stimulus
  - Stream of independently randomized transactions will not exercise interesting DUT behaviors
- Scenarios provide this
  - and offer a natural way to express complex stimulus

Outline

- Introduction: motivation for scenarios
- The VMM Standard Library scenario generator
- Motivation for hierarchical scenarios
- Prototype VMM-compatible hierarchical scenarios
- Further opportunities and future work
- Summary

The VMM Scenario Generator

- Part of VMM Standard Library
- Most of the work is done for you automatically
  - by macro `` `vmm_scenario_gen() ``
- Multiple levels of customization are possible
  - Simple example on the next few slides
  - Fuller description in section 3 of the printed paper

Scenario Generator Macro

```
class XYZ_data extends vmm_data;
  ...
endclass : XYZ_data
`vmm_channel(XYZ_data)
`vmm_atomic_gen(XYZ_data, "XYZ atomic")
`vmm_scenario_gen(XYZ_data, "XYZ scenario")
```

Creates...

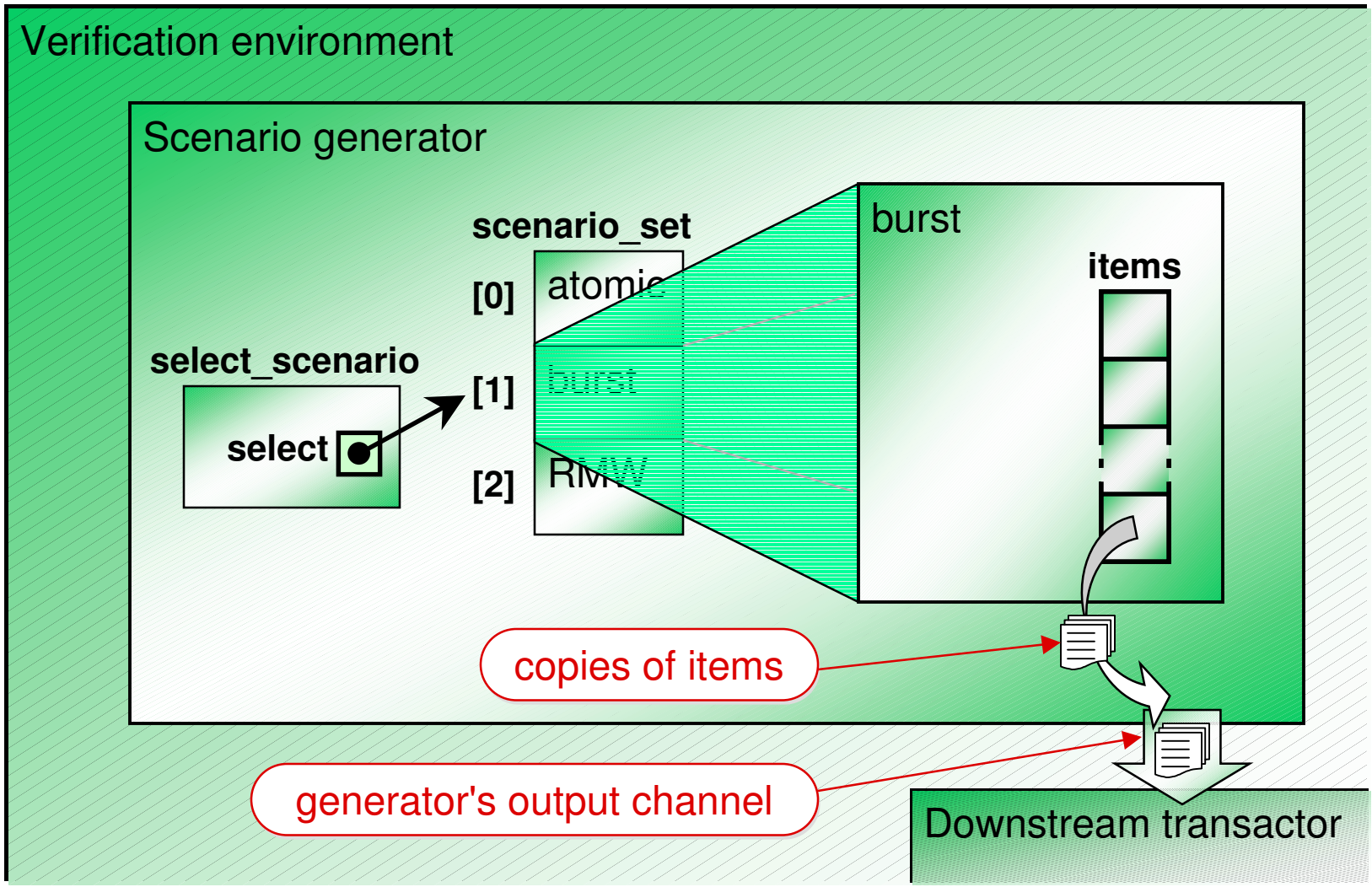
base for your scenarios — `class XYZ_data_scenario`

predefined simple scenario — `class XYZ_data_atomic_scenario`

scenario chooser — `class XYZ_data_scenario_election`

generator (transactor) — `class XYZ_data_scenario_gen`

Scenario Generator Operation



The Scenario Set

within scenario-generator class

- Set of scenarios available for selection

```
data_class_scenario scenario_set[$];
```

SystemVerilog *queue*

- By default, contains just one atomic scenario object
- Create instances of your own scenarios, add them to this queue
  - It's an array of factory objects

- By default, scenario selector class chooses scenarios in cyclic order
  - Make the choice random:

```
gen.select_scenario.round_robin.constraint_mode(0);
```

your scenario generator instance

What's in a Scenario?

```
rand data_class items[$];
```

Array of transaction objects

- You apply constraints to the `items[]` array (in a derived class):

```
class apb_RMW_scenario extends apb_data_scenario;
  ...
  constraint read_modify_write {
    length == 2;
    items[0].kind == apb_data::READ;
    items[1].kind == apb_data::WRITE;
    items[0].address == items[1].address;
  }
```

desired number of entries in `items[]`

- Generator chooses a scenario from scenario_set, then randomizes it

Multiple Scenarios in One Subclass

```
rand int unsigned scenario_kind;
```

choice within any subclass

```
int APB_RMW, APB_BURST;
constraint read_modify_write {
  if (scenario_kind == APB_RMW) {
    length == 2;
    ...
  }
}
constraint burst {
  if (scenario_kind == APB_BURST) {
    length inside { 2, 4, 8, 16 };
    ...
  }
}
```

scenario kind identifiers

randomized property of base class

- How do we set up the kind-identifiers?

Creating Scenario Identifiers

- Identifier values created by scenario's constructor

```
function new();
  super.new();
  APB_RMW = define_scenario("APB R-M-W", 2);
  APB_BURST = define_scenario("APB burst", 16);
endfunction
```

Establishes set of possible values for `scenario_kind`

Largest length each scenario can take

- Unique named values automatically maintained, even if there are intermediate derived classes
- Low cost - scenario is constructed only once per generator

You MUST define at least one scenario kind

The **apply** Method

- After the scenario has been randomized, its items are ready for delivery to the output channel

- Generator calls scenario's apply method to do this
  - Default implementation (paraphrase):

```
virtual task apply (
    apb_data_channel channel,
    ref int unsigned n_insts);
  foreach (items[i])
    channel.put(items[i].copy());
  n_insts = items[i].size();
endtask
```

reference to generator's output channel

allows generator to count transactions

Override `apply` to create procedural scenarios

Outline

- Introduction: motivation for scenarios
- The VMM Standard Library scenario generator
- Motivation for hierarchical scenarios
- Prototype VMM-compatible hierarchical scenarios
- Further opportunities and future work
- Summary

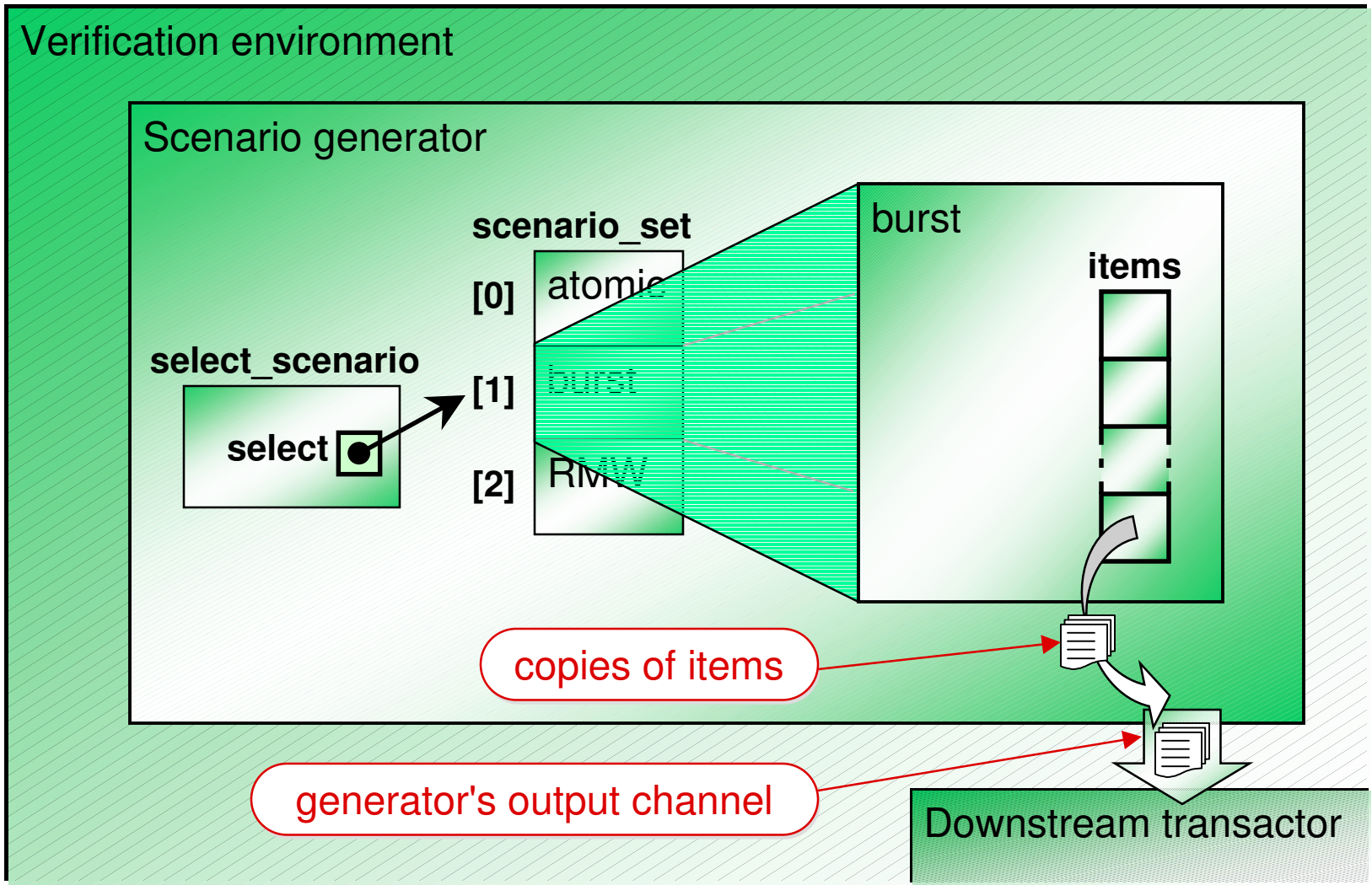
Motivation for hierarchical scenarios

- Compose scenarios from existing scenarios
  - Saves coding effort
  - Facilitates re-use
- Naturally express nested activities:
  - layered protocols
  - instruction streams
  - high-level usage patterns

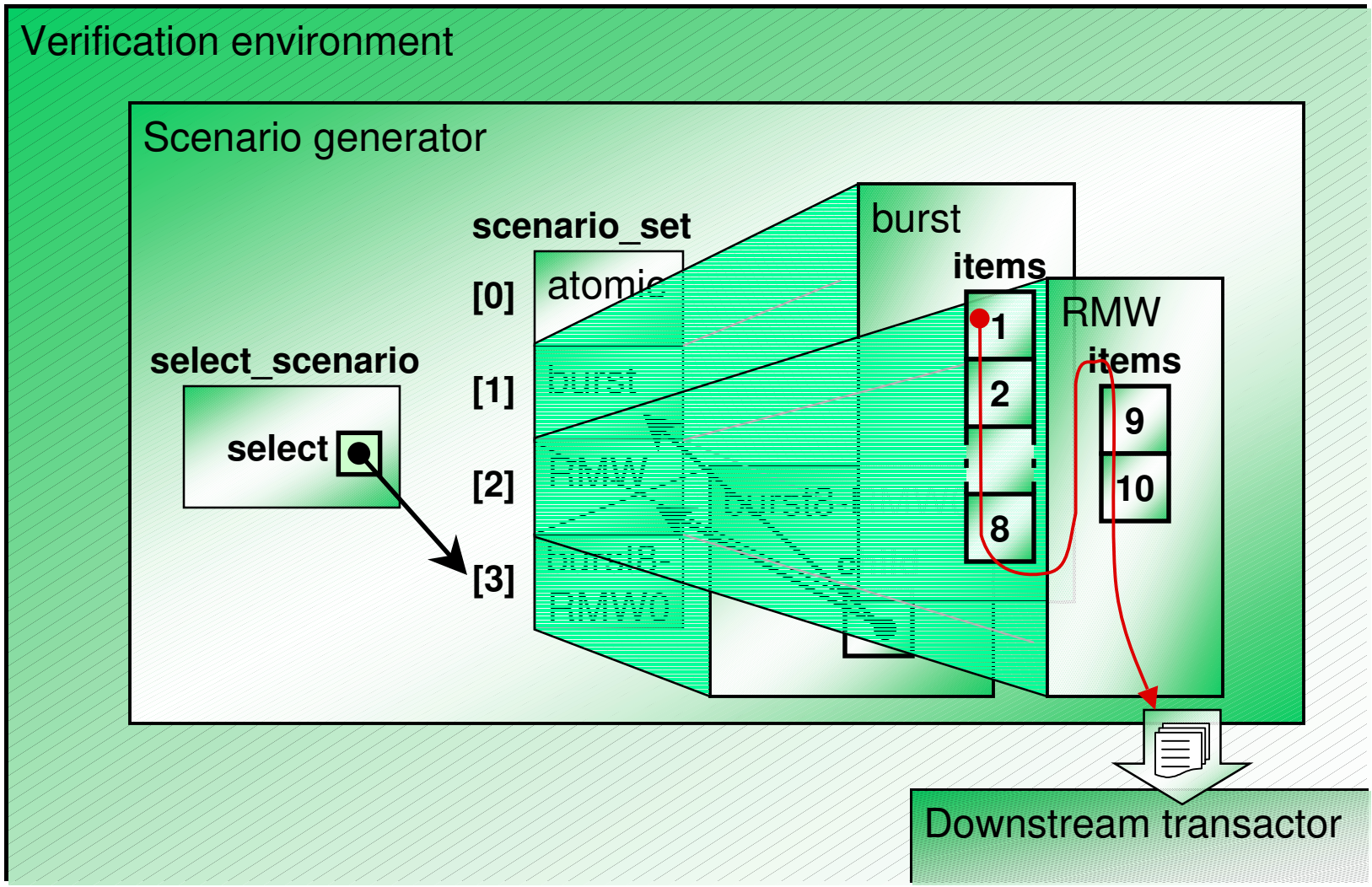
Outline

- Introduction: motivation for scenarios
- The VMM Standard Library scenario generator
- Motivation for hierarchical scenarios
- Prototype VMM-compatible hierarchical scenarios
- Further opportunities and future work
- Summary

Scenario Generator Operation (review)



Hierarchical Scenario

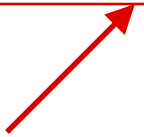


Problems to solve

- Hierarchical scenario must do most things that the generator currently does...
  - choose a scenario
  - randomize it
  - deliver its items to the output channel
- Randomization may need additional constraints
  - Awkward to write additional derived classes
  - Need inline constraints like randomize...with {...}
- Allow for multiple levels of scenario hierarchy

Structuring a solution

- Hierarchical scenario must do most things that the generator currently does...
  - choose a scenario — use the generator's scenario selector
  - randomize it — this.randomize
  - deliver its items to the output channel — this.apply



- **`perform`** method of hierarchical scenario base class
  - `apply` method may choose/`perform` child scenario(s)

Outline **apply** Method of Hierarchical Scenario

- For each sub-scenario:
  - ask generator to choose from its scenario_set
  - ask the chosen sub-scenario to perform itself

```
class apb_multi_scenario extends apb_data_HS;
  ...
  virtual task apply (...);
    apb_data_HS child;
    repeat (number_of_scenarios) begin
      child = parent_generator.choose(...);
      child.perform(...);
      n_insts++;
    end
  endtask
  ...
endclass
```

randomized property of this scenario?

customize choice

customize randomization

```
class apb_data_HS extends apb_data_scenario;
  apb_data_HS_gen parent_generator;
  function new (apb_data_HS_gen pg);
    super.new();
    parent_generator = pg;
  endfunction
  ...
```

scenario has reference to its generator

The `perform` Method

- With no *constrainer*, mimics generator's behavior
- With a constrainer, custom randomization is easy to achieve

```
class apb_data_HS extends apb_data_scenario;
  ...
  virtual task perform (
      apb_data_channel  channel,
      ref int unsigned  n_insts,
      apb_data_HS_constrainer c = null );
    if (c == null)
      this.randomize();
    else
      c.do_randomize( this );
    this.apply( channel, n_insts );
  endtask
```

users don't need to override this method in a derived scenario

strategy class to customize randomization

Custom constrainer

- Provides a convenient way to do `randomize() with {...}`

```
class apb_burst8 extends apb_data_HS_constrainer;
  virtual function void do_randomize( apb_data_scenario it );
    apb_RMW_burst_scenario s;
    assert ( $cast(s, it) ) else $error("Bad constrainer");
    s.randomize() with {
      scenario_kind == APB_BURST;
      length == 8;
    );
  endfunction
endclass
```

Only one function to override!

```
virtual task perform (
    apb_data_channel  channel,
    ref int unsigned  n_insts,
    apb_data_HS_constrainer c = null );
  if (c == null)
    this.randomize();
  else
    c.do_randomize( this );
  this.apply( channel, n_insts );
endtask
```

scenario

Choosing a Scenario

- Expose generator's choice mechanism

```
class apb_data_HS_gen extends apb_data_scenario_gen;
  virtual function apb_data_scenario choose
        ( apb_data_HS_chooser c = null );

    if (c == null)
      this.select_scenario.randomize();
    else
      c.choose(this.select_scenario);

    return this.scenario_set[this.select_scenario.select];
  endfunction
  ...
endclass
```

scenario calls this to choose a child scenario

allows user control of choice strategy

default: same as standard generator

chooser class customizes choice

return a reference to chosen member of scenario set

A Simple User-written Scenario

- An 8-word burst write followed by a read-modify-write to address 0

```
virtual task apply (
    apb_data_channel channel ,
    ref int unsigned n_insts );

  special_chooser ch = new;
  apb_burst_W8_constrainer c_w8 = new;
  apb_RMW_addr0_constrainer c_rmw0 = new;
  apb_data_HS child = parent_generator.choose(ch);
  child.perform(channel, n_insts, c_w8);
  child.perform(channel, n_insts, c_rmw0);

endtask
```

choose a suitable scenario

randomize and apply

randomize and apply

custom constraints

Outline

- Introduction: motivation for scenarios
- The VMM Standard Library scenario generator
- Motivation for hierarchical scenarios
- Prototype VMM-compatible hierarchical scenarios
- Further opportunities and future work
- Summary

Future work

- Better integration with existing VMM infrastructure
  - Notifications, callbacks, logging
- Macros for convenience
  - low overhead on top of existing VMM scenarios
- Re-entrancy
  - define scenarios in terms of themselves with different constraints
- Debug and visibility

Further possibilities

- Co-ordinated parallel scenarios
  - Each scenario can now choose to which channel it sends its sub-scenarios (argument to `perform`)
  - A scenario can generate multiple scenarios in parallel

```
virtual task apply (...);
  ...
  apb_data_HS s_main = parent_generator.choose(...);
  apb_data_HS s_intrpt = parent_generator.choose(...);
  fork
    s_main.perform(main_gen.out_chan, ...);
    s_intrpt.perform(irq_gen.out_chan, ...);
  join
endtask
```

Outline

- Introduction: motivation for scenarios
- The VMM Standard Library scenario generator
- Motivation for hierarchical scenarios
- Prototype VMM-compatible hierarchical scenarios
- Further opportunities and future work
- Summary

Summary

- Standard VMM scenarios are easy to use
  - but customization can be laborious
- Hierarchical scenario generator is possible
  - useful results from our prototype
  - more work required to integrate fully with VMM
  - some known limitations (not re-entrant)
- Planning for extensibility and re-use is challenging
  - but inescapable

Thanks for your attention!

Questions?

Jonathan Bromley

Doulos Ltd, Ringwood, UK

jonathan.bromley@doulos.com