2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

IEEE-Compatible UVM Reference Implementation and Verification Components

Justin Refice – Nvidia
Mark Strickland – Cisco Systems
Mark Peryer – Mentor, a Siemens Business
Uwe Simm – Cadence Design Systems
Srivatsa Vasudevan – Synopsys

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Agenda

- Introduction – what and why (Mark@Cisco)
- Information for new users and migrators from pre-1.2 (Mark@Cisco)
- Highlights, detail and examples
  - UVM Object, UVM Policies (Mark@Mentor)
  - Abstract Factory, Deferred Initialization, Dynamic UVM_Reg mapping (Uwe@Cadence)
  - Config, Callbacks, Reporting, Sequences (Sri@Synopsys)
- Future

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Accellera UVM Working Group

- Accellera has initiatives focusing on Verification challenges
  - UVM, SystemC, Multi-Language
- The UVM WG consists of volunteers from both tool providers and end user companies
- The UVM WG used to do all development associated with UVM (the most recent release is UVM 1.2) but now works in conjunction with IEEE 1800.2

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

IEEE-Compatible UVM Reference Implementation and Verification Components

Introduction, New Users and Migrators from Pre-1.2

Mark Strickland – Cisco Systems

Re-Introduction to 1800.2

- UVM WG now provides just an implementation of IEEE standard 1800.2
  - 1800.2-2017 Standard is available at no cost courtesy of the IEEE Get Program™ and Accellera

Deliverable	UVM-1.2 Provider	1800.2 Provider
Standard (LRM)	Accellera UVM	IEEE
Implementation (Library)	Accellera UVM	Accellera UVM
User's Guide / Examples	Accellera UVM	3rd parties

  - Note that implementations can support a superset of the API described in the standard, and the Accellera implementation does that

1800.2 Implementation Steps

- Process used by the UVM WG:
  - Start with UVM1.2 library
  - Remove all code under deprecation in UVM1.2
  - Add all 1800.2 API that was not present in 1.2 and have the library use that new API when applicable
  - Modify (few) API that were present in 1.2 but now have different function or signature
  - Deprecate 1.2 API for which 1800.2 provides a recommended alternative
  - Fix some implementation bugs

1800.2 Library Compared to 1.2

- New API enables some new functionality for users
  - Could have been impossible with 1.2 or could have been possible only with undocumented API
- Alternate API provides a new (recommended) way to implement existing functionality
- A few 1.2 API are incompatible with 1800.2 and will be unavailable

Superset of 1800.2 API

- Compliant implementations must implement all the documented API in the standard, but they may also implement other API

- The Accellera library includes the following categories of extra API:
  - API used within the library, not intended to be public
  - Debug aids
  - API that may be submitted for consideration for future 1800.2 versions
  - (when a define is set) UVM1.2 API that does not conflict with 1800.2 API

API Supporting Transition from UVM1.2

- The 1800.2 standard replaced some UVM1.2 API with different API
  - Typically replacing direct field access with accessor methods

- When possible, the 1.2 API is provided in the library when `` `UVM_ENABLE_DEPRECATED_API `` is defined
  - Only possible if the 1.2 API didn't contradict 1800.2
  - When defined, *both* the 1.2 and 1800.2 APIs are available
  - When not defined, user code *must not* reference the 1.2 API, or compile will fail

BY DEFAULT, `` `UVM_ENABLE_DEPRECATED_API `` **IS NOT DEFINED!!!**

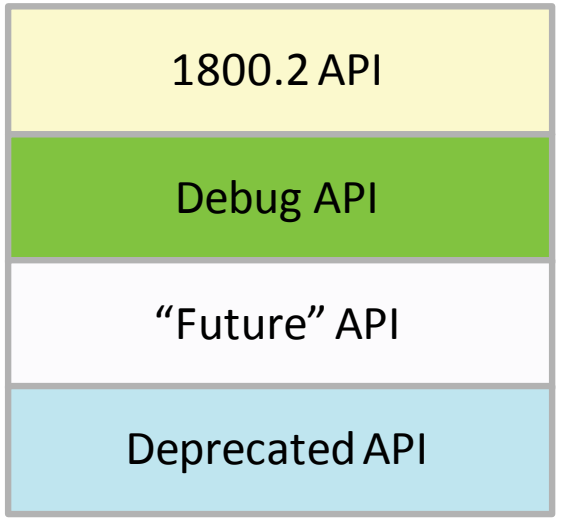
Sometimes Deprecation Not Possible

- In a small number of situations, old code may not compile even with the deprecated flag
- Example: `uvm_packer::use_metadata`
  - Not in 1800.2-2017, `uvm_policy` features support the intent
  - Would require a default value swap to be compatible with `pack/unpack_object`
  - Simply not compatible with `pack/unpack_string`

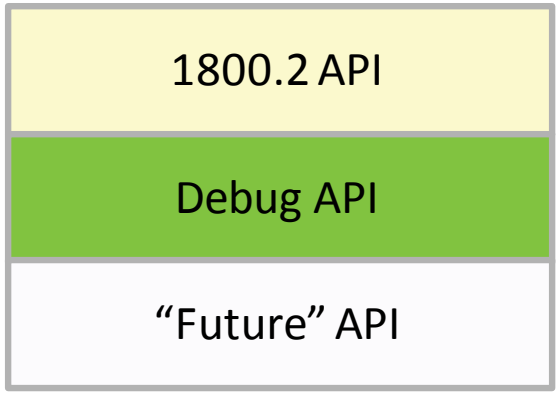
Library Always Supports All 1800.2

- If UVM1.2 API was in conflict with the 1800.2 API, it was removed
- Even with `UVM_ENABLE_DEPRECATED_API defined, the library will support all of the 1800.2 API syntax and semantics

With `UVM_ENABLE_DEPRECATED_API

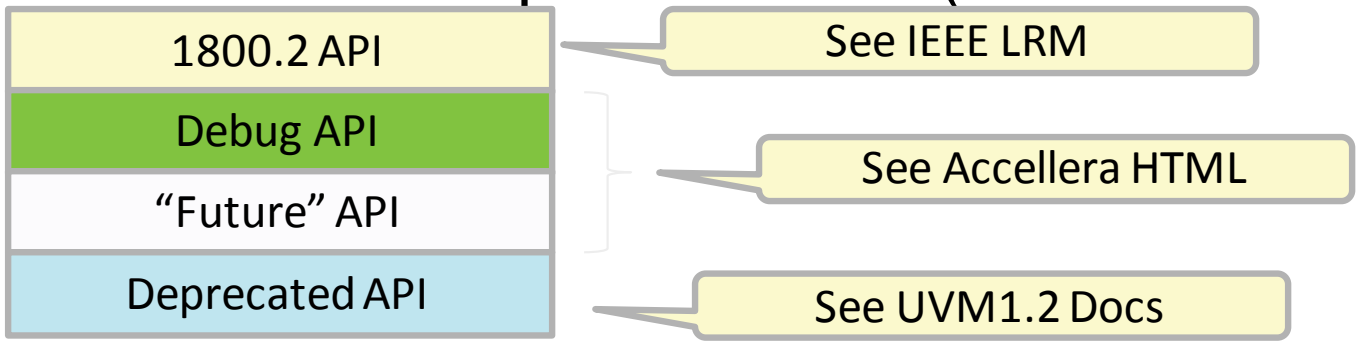


Without `UVM_ENABLE_DEPRECATED_API



What About Documentation?

- Accellera will provide HTML documentation covering:
  - Extra API that is intended to be used by the end user
  - Supplemental information for 1800.2 API that clarifies behavior beyond what the 1800.2 standard says
- Accellera documentation will *not* include:
  - Any information included in IEEE 1800.2 LRM
  - Any information on deprecated API (described in UVM1.2 Docs)



Does User Code Rely on Extra API?

- If you use *only* the API that is documented in the standard, then your code can switch to another implementation without a problem.
- Accellera ***does not provide a compliance check*** to ensure extra API is not used.
  - Compile errors if user code relies on the deprecated UVM1.2 API if the user does not set the define to explicitly enable that API.
  - The implementation of all 1800.2 API is tagged with a comment that could be used by lint tools to distinguish the use of 1800.2 API and other API.

```
// @uvm-ieee 1800.2-2017 auto 5.3.1
virtual class uvm_object extends uvm_void;
```

Recommended Steps for Updating

1. Compile/Run against UVM1.2 with `` `UVM_NO_DEPRECATED `` defined
   - Not the subject of this tutorial

2. Compile/Run against 1800.2-2017 with `` `UVM_ENABLE_DEPRECATED_API `` defined
   - Any new compile failures are the result of incompatible 1.2 APIs

3. Compile/Run against 1800.2-2017 without `` `UVM_ENABLE_DEPRECATED_API ``
   - Any new compile failures are the result of deprecated 1.2 APIs
   - Linting tools can ease this transition

How Library Will Be Released

- Early Adopters Release
  - All the implementation is complete
  - Accellera regression passes
  - Documentation of extra API is incomplete
- Full 1800.2-2017 Release
  - Documentation is complete

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

THANK YOU!

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

IEEE-Compatible UVM Reference Implementation and Verification Components

Objects and Policies

Mark Peryer – Mentor, a Siemens Business

Changes to `uvm_object`

- New UVM seeding methods
  - These interact with the seed setting in `uvm_coreservice_t`
  - If seeding is enabled:
    - Random seeding is based on the full name rather than object creation order

```
static function bit get_uvm_seeding();
static function void set_uvm_seeding (bit enable);
```

- New methods for configuration and policy interactions

```
virtual function void do_execute_op( uvm_field_op op );
virtual function void set_local ( uvm_resource_base rsrc );
```

About `do_execute_op()`

- Additional call-back to add flexibility in field operations
  - Configuration and policy interaction
  - It takes an `uvm_field_op` object as its argument

- `uvm_field_op` extends `uvm_object`, and contains:
  - A `uvm_policy` object (default **`null`**)
    - e.g. `uvm_printer`
  - A field flag to indicate operation type
    - Built in: `UVM_COPY`, `UVM_COMPARE`, `UVM_PRINT`, `UVM_RECORD`, `UVM_PACK`, `UVM_UNPACK`, `UVM_SET`
    - User defined flag fields can be used for other field operations
  - A right hand side ("rhs") `uvm_object`
    - Used in copy, compare and set operations by default
    - Defaults to **`null`** in other operations

Configuration

- Configuration can be done as before:
  - Using field macros for object or component members
  - Via the `uvm_config_db`
  - Explicitly via assignment from a configuration object
- However:
  - The field macros implement default `do_execute_op()` code
    - Setting the `uvm_field_op` field flag to `UVM_SET`
  - If you use field macros
    - You will need to take account of this in any `do_execute_op()` overlays
  - Otherwise you could now implement configuration via `do_execute_op()`

Policy Classes in 1800.2

- Changes between 1800.2 and UVM 1.2
- Overall use model changes
- Some example applications

Changes Between UVM 1.2 and 1800.2

- All object field methods now have a policy
  - i.e. `copy()`, `compare()`, `pack/unpack()`, `record()`, `print()`
  - Addition of `uvm_copier` policy object
- All policy classes now extend from a common `uvm_policy` class
- The `uvm_policy` class adds some base functionality:
  - The policy state can be reset using the `flush()` method
  - Object specific extensions can be added
  - Keeps a stack of active objects
  - Has a recursion state property

Policy Extensions

- Each policy can have extensions
  - Similar to the TLM2 generic payload
- The extensions are indexed by a `uvm_object`'s type
  - Only one extension of each type can exist
- Methods:
  - `set/get_extension()`
  - `extension_exists()`
  - `clear_extension()`
  - `clear_extensions()`
- These can be used to customise how an object executes a policy

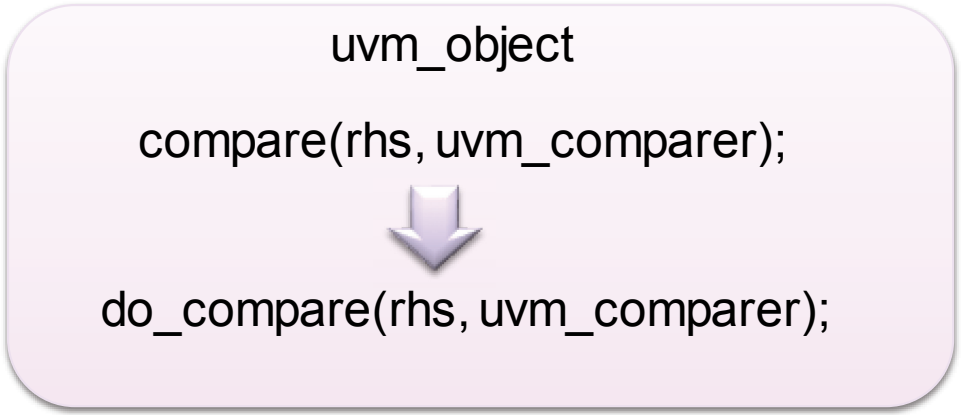
Use Model Changes

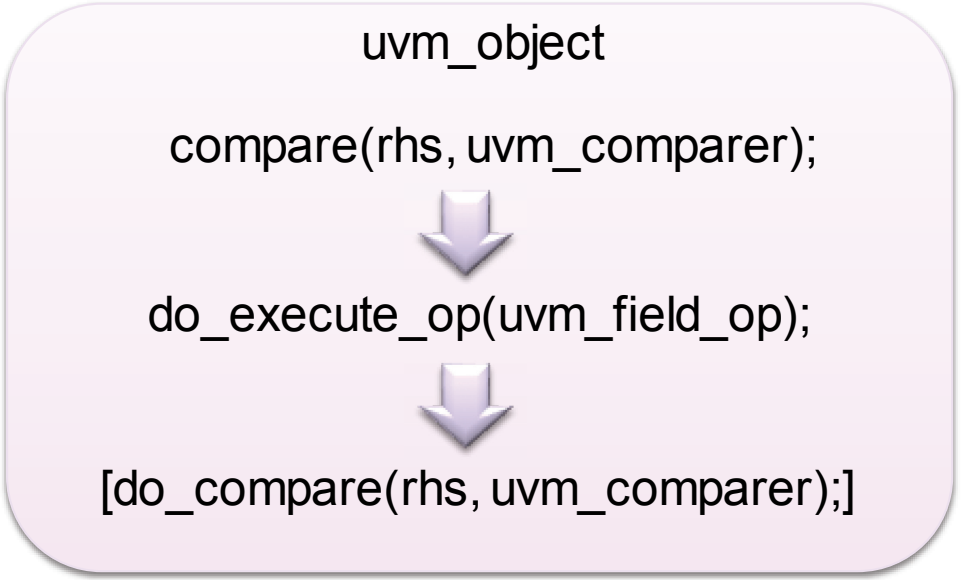
- Before:
  - Policy object provided methods that allowed you to perform field specific operations
  - Object's `do_xxx()` method could call a policy `xxx_<field>()` method to do *xxx*

```
function bit do_compare(uvm_object rhs, uvm_comparer comparer);
  my_type _rhs;
  do_compare = super.compare(rhs, comparer)) ;
  if(!$cast(_rhs, rhs)) return 0;
  do_compare &= comparer.compare_field_int("f1", f1, _rhs.f1);
endfunction
```

- With 1800.2:
  - Can still work as before – i.e. no code changes == no differences
  - Extension objects can be used to customize field operations
  - The `do_execute_op()` call-back can be used to process objects before `do_xxx()`

Change In Object Behaviour





UVM 1.2

- Hardwired behaviour
- `compare_object()` called `do_compare()`
- The content of `do_compare()` is fixed

UVM 1800.2

- Adds flexibility
- `do_execute_op` is an optional call-back
  - Can set a flag to ignore `do_<policy>()` method
- Objects can interact with policy extensions

Putting This All Together

- Extra flexibility added via policy extensions
  - A way to change the behaviour of `do_xxx()` methods
- Policy object operation methods now call `do_execute_op()`
  - This gives another layer of customisation
- For instance:
  - You could have two ways defined to carry out a policy
    - One in `do_execute_op()` and another in `do_<policy>()`
    - Which one is used is dependent on a policy extension or another object flag
- Some use model examples follow…

Copier Example: Copy Alternatives

The Example Object:

```
class bus_seq_item extends uvm_sequence_item;

`uvm_object_utils(bus_seq_item)

rand bit[31:0] address;
rand bit read;
rand bit[3:0] cache_ops;
rand bit[1:0] burst_type;
rand bit[7:0] burst_length;
rand bit[5:0] protection_attribute;
rand bit[3:0] qos;
rand bit[31:0] write_data[];

bit[31:0] read_data[];
bit[1:0] bus_resp;

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

extern function void do_execute_op(uvm_field_op op);
extern function void do_copy(uvm_object rhs);

endclass
```

Checks for policy extension, if found, do_copy() never called

```
function void bus_seq_item::do_execute_op(uvm_field_op op);
  bus_seq_item rhs_;
  uvm_copy_policy copier;
  bus_seq_copy_filter copy_filter;

  if(op.get_name() == "copy") begin
    copier = op.get_policy();
    if(copier.extension_exists(bus_seq_copy_filter::type_id))
    begin
      op.disable_user_hook(); // do_copy disabled
      // Reduced copy implementation:
      if($cast(rhs_, op.get_rhs()) begin
        this.address = rhs_.address;
        this.read = rhs_.read;
        this.resp = rhs_.resp;
      end
    end
  end
  // ...
endfunction
```

```
// Full implementation of the copy policy
function void bus_seq_item::do_copy(uvm_object rhs);
  bus_seq_item rhs_;

  super.do_copy(rhs_);
  if($cast(rhs_, rhs)) begin
    this.address = rhs_.address;
    this.read = rhs_.read;
    this.cache_ops = rhs_.cache_ops;
    this.burst_type = rhs_.burst_type;
    this.protection_attribute = rhs_.protection_attribute;
    this.qos = rhs_.qos;
    this.write_data = rhs_.write_data;
    this.read_data = rhs_.read_data;
    this.resp = rhs_.resp;
  end
endfunction
```

Some Notes About `copy()`

- The `copy()` method signature has changed:

```
function void copy(uvm_object rhs, uvm_copier copier = null);
```

  - This is backwards compatible
    - `copy()` is not virtual
    - `copier` has a default value

- The `do_copy()` method signature remains the same:

```
virtual function void do_copy(uvm_object rhs);
```

Some Notes About `uvm_copier`

- The copier has a `copy_object()` method which is applied to the lhs
- When copying hierarchical objects policy defines recursion policy:
  - UVM_DEEP (copier.copy_object(tgt, src)) ** Default
  - UVM_SHALLOW (tgt.field = src.field)
- Unlike other policies:
  - Apart from `copy_object()`, the copier policy has no `copy_xxx()` methods
  - The copier must be retrieved via `get_active_policy()`
  - `UVM_REFERENCE` is an illegal recursion policy for `copy_object()`

    ** An error will be flagged for object copies (uvm_object handle)

Recorder Example

- In debug situations we often need to sort the wood from the trees
- We might want to control the level of detail recorded for a transaction
  - More detail at the point of failure
  - Less detail at other times
- This has a potential impact on GUI real estate and performance
- The following example shows how the 1800.2 policy updates could be used to achieve this

Recorder Example Detail

Policy extension – Txn detail filter

```
// Transaction recording detail enum
typedef enum {LITE, MEDIUM, FULL} detail_e;

//
// Transaction recording detail filter
//
class bus_recording_detail extends uvm_object;

`uvm_object_utils(bus_recording_detail)

local detail_e recording_detail;

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

// set/get_recording_detail(); Accessors

endclass
```

API sequence to:

- Create recording detail extension
- Set the transaction recording detail
- Apply it to the current recorder policy

```
class set_detail_sequence extends uvm_sequence #(bus_seq_item);

bus_recording_detail detail_extension;
detail_e recording_detail;

task body;
  uvm_recorder recorder =
       uvm_recorder::get_recorder_from_handle(get_tr_handle());

  detail_extension =
       bus_recording_detail::type_id::create("detail_extension");

  detail_extension.set_recording_detail(recording_detail);
  recorder.set_extension(detail_extension);
endtask

endclass
```

Applying Detail & do_record()

```
class uber_sequence extends uvm_sequence #(bus_seq_item);

set_detail_sequence set_detail;
transfer_data_sequence write;
transfer_data_sequence read;

task body;
  set_detail =
          set_detail_sequence::type_id::create("set_detail");
  write = transfer_data_sequence::type_id::create("write");
  read = transfer_data_sequence::type_id::create("read");

  set_detail.set_recording_detail(MEDIUM);
  set_detail.start(get_sequencer());
  repeat(20) begin
    assert(write.randomize() with {read == 0;});
    write.start(get_sequencer());
    assert(read.randomize() with {read == 1;});
    read.start(get_sequencer());
  end
  set_detail.set_recording_detail(FULL);
  set_detail.start(get_sequencer());
  assert(read.randomize() with {read == 1;});
  read.start(get_sequencer());

endtask
endclass
```

```
function void bus_seq_item::do_record(uvm_recorder recorder);
  bus_recording_detail detail_policy;
  detail_e detail_level;

  super.do_record(recorder);

  if(recorder.extension_exists(bus_recording_detail::type_id))
    begin
      $cast(detail_policy,
       recorder.get_extension(bus_recording_detail::get_type()));
      detail_level = detail_policy.get_recording_detail();
    end

  // LITE recording default
  `uvm_record_int("ADDR", address, 32, UVM_HEX)
  `uvm_record_int("READ", read, 1, UVM_BIN)
  `uvm_record_int("RESP", bus_resp, 2, UVM_BIN)
  // Medium recording detail
  if((detail_level == MEDIUM) | (detail_level == FULL))
    begin
      `uvm_record_int("Burst Length", burst_length, 8, UVM_DEC)
      `uvm_record_int("Burst type", burst_length, 2, UVM_HEX)
      if(read == 1) begin
        foreach(read_data[i]) begin
        `uvm_record_int($sformatf("read_data[%0d]", i),
                        read_data[i], 32, UVM_HEX)
      end
    end
    ...
```

Comparer Example

- In scoreboards there is a need to compare objects of differing types
- The following example shows how to add a policy extension to achieve this
  - Comparing a `bus_seq_item` against a `uvm_tlm_generic_payload`

Policy Extension Comparer

```
class compare_generic_payload extends uvm_object;

  function bit compare_against_bus_item(bus_seq_item bsi, uvm_tlm_generic_payload gp);
    byte unsigned data[];
    if(gp == null) begin
      return 0;
    end
    compare_against_bus_item = (bsi.address == gp.m_address[31:0]);
    if(gp.is_read()) begin
      compare_against_bus_item = compare_against_bus_item & (bsi.read == 1);
      compare_against_bus_item = compare_against_bus_item & ((bsi.read_data.size())*4 == gp.get_data_length());
      data = convert_bus_item_data(bsi.read_data);
      compare_against_bus_item = compare_against_bus_item & (data == gp.get_data());
    end
    else if(gp.is_write()) begin
      compare_against_bus_item = compare_against_bus_item & (bsi.read == 0);
      compare_against_bus_item = compare_against_bus_item & ((bsi.write_data.size())*4 == gp.get_data_length());
      data = convert_bus_item_data(bsi.write_data);
      compare_against_bus_item = compare_against_bus_item & (data == gp.get_data());
    end
    else begin
      return 0;
    end
  endfunction

endclass
```

Example do_execute_op()

```
function void bus_seq_item::do_execute_op(uvm_field_op op);
  uvm_comparer_policy comparer;
  compare_generic_payload comp_gc;

  if(op.get_name() == "compare") begin
    comparer = op.get_policy();
    if(comparer.extension_exists(compare_generic_payload::type_id
      uvm_tlm_generic_payload gp;

      comp_gc = comparer.get_extension(compare_generic_payload::t
      $cast(gp, comparer.get_rhs());

      if(!comp_gc.compare_against_bus_item(this, gp)) begin
        comparer.result++; // If the comparison fails
      end
      op.disable_user_hook();
    end
  end

endfunction
```

Note:

`do_compare()` is implemented as normal i.e. comparing a `bus_seq_item` to another `bus_seq_item`

```
class gp_scoreboard extends uvm_component;

uvm_comparer cmpr;
compare_generic_payload compare_gp;

bus_seq_item bus_item;
uvm_tlm_generic_payload gp_item;

uvm_tlm_analysis_fifo #(bus_seq_item) bus_item_fifo;
uvm_tlm_analysis_fifo #(uvm_tlm_generic_payload) gp_fifo;

int comparison_errors;

function void build_phase(uvm_phase phase);
  cmpr = new("cmpr");
  compare_gp = compare_generic_payload::type_id::create("compare_gp");
  cmpr.set_extension(compare_gp);
  bus_item_fifo = new("bus_item_fifo", this);
  gp_fifo = new("gp_fifo", this);
end

task run_phase(uvm_phase phase);
  forever begin
    bus_item_fifo.get(bus_item);
    gp_fifo.get(gp_item);
    if(!cmpr.compare_object("bus_seq_item <=> uvm_tlm_generic_payload
miscomparison", bus_item, gp_item)) begin
      comparison_errors++;
    end
  end
endtask

endclass
```

Scoreboard and do_execute_op()

```
class gp_scoreboard extends uvm_component;

uvm_comparer cmpr;
compare_generic_payload compare_gp;

bus_seq_item bus_item;
uvm_tlm_generic_payload gp_item;

uvm_tlm_analysis_fifo #(bus_seq_item) bus_item_fifo;
uvm_tlm_analysis_fifo #(uvm_tlm_generic_payload)
gp_fifo;

int comparison_errors;

extern function void build_phase(uvm_phase phase);
extern task run_phase(uvm_phase phase);

endclass
```

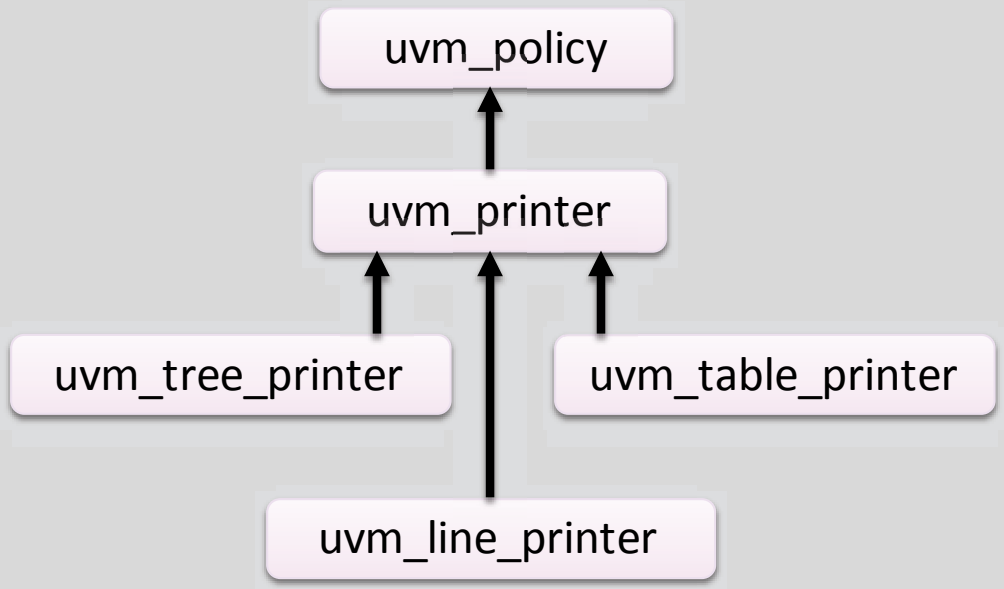
```
function void gp_scoreboard::build_phase(
                                    uvm_phase phase);
  cmpr = new("cmpr");
  compare_gp = new("compare_gp");
  cmpr.set_extension(compare_gp);
  bus_item_fifo = new("bus_item_fifo", this);
  gp_fifo = new("gp_fifo", this);
endfunction

task gp_scoreboard::run_phase(uvm_phase phase);
  forever begin
    bus_item_fifo.get(bus_item);
    gp_fifo.get(gp_item);
    if(!bus_item.compare(gp_item, cmpr)) begin
      comparison_errors++;
    end
  end
endtask
```

pack()/unpack()

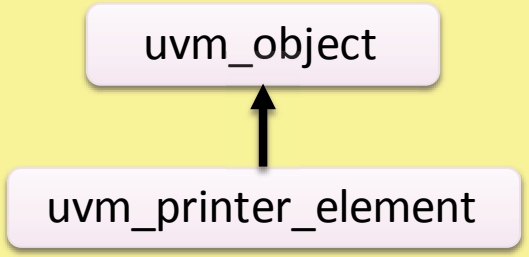
- `pack()` converts object fields to bits, `unpack()` vice versa
- Useful for moving objects between languages or platforms
  - SystemC, SV UVM, hardware transactors, etc
- Only works if the `pack()` and `unpack()` methods mirror each other
- 1800.2 relaxes the method definitions
  - In particular for `pack_string()` & `unpack_string()`
  - Enables alternative `uvm_packer` implementations

UVM Printer Policies

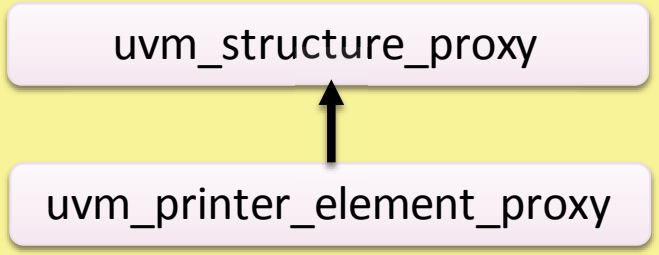
- The standard UVM printers now extend from `uvm_policy`.
- Now uses `uvm_printer_elements`, but same functionality



- Two new classes:
  - `uvm_printer_element`
  - `uvm_printer_element_proxy`
- These enable printer extensions with arbitrary hierarchical type capabilities



Printer line object



Object Hierarchy Visitor

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

For Example: A JSON Printer

- Many post-processing visualization tools use JSON as a db file format
  - Other possibilities include XML, CSV, etc.
- JSON and XML have nested description tags to document hierarchy
- The uvm_printer is implemented with an element stack to enable hierarchy
- In this example we will create a transaction stream JSON file

The Transaction – Sequence Item

```
class bus_extension extends uvm_object;

`uvm_object_utils(bus_extension)

rand bit[3:0] security;
rand bit[3:0] cache;
rand bit[2:0] coherency;

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

function void do_print(uvm_printer printer);
  printer.print_field("security", security, 4, UVM_HEX);
  printer.print_field("cache", security, 4, UVM_HEX);
  printer.print_field("coherency", security, 3, UVM_HEX);
endfunction

endclass
```

Bus extension object, nested inside the sequence item, allowing separate randomization during generation.

print_array_header(), print_object() used by uvm_printers to push onto element stack

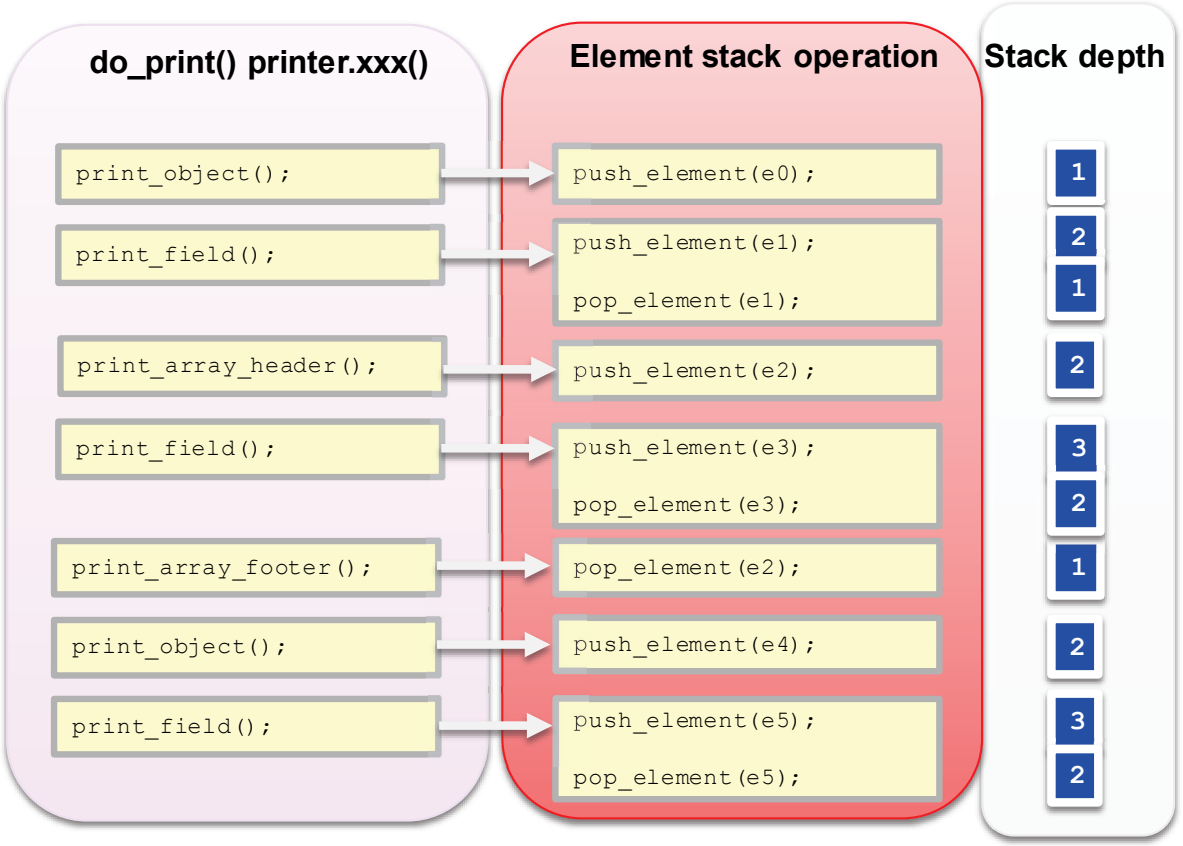
```
class bus_item extends uvm_sequence_item;

rand bit[31:0] address;
rand bit write;
rand bit[31:0] write_data[];
rand int burst_length;
rand bus_extension extension;

bit[31:0] read_data[];
bit read_resp[];
bit write_resp;

function void do_print(uvm_printer printer);
  printer.print_time("start_time",get_start_time(), ":");
  printer.print_field("address", address, 32, UVM_HEX);
  printer.print_field("write", write, 1, UVM_BIN);
  printer.print_field("burst_length", burst_length, 32, UVM_DECIMAL);
  if(write == 1) begin
    printer.print_array_header("write_burst", burst_length);
    foreach(write_data[i]) begin
      printer.print_field($sformatf("wdata[%0d"], i), wdata[i], 32, UVM_HEX);
    end
    printer.print_array_footer();
    printer.print_field("write_resp", write_resp, 1, UVM_BIN);
  end
  else begin
    printer.print_array_header("read_burst", burst_length);
    foreach(read_data[i]) begin
      printer.print_field($sformatf("rdata[%0d"], i), rdata[i], 32, UVM_HEX);
      printer.print_field($sformatf("resp[%0d"], i), rresp[i], 1, UVM_BIN);
    end
    printer.print_array_footer();
  end
  printer.print_object(extension);
endfunction

endclass
```

The JSON Printer – Part 1

```
class json_printer extends uvm_printer;

int depth;
string indent = "  "; // Each hierarchical element needs an indent

// Element stack handling
extern function string emit_element(int depth,
                                    uvm_printer_element element);
extern function string open_hierarchy(int depth, string name,
                                      string type);
extern function string close_hierarchy(int depth);
extern function string emit_nh_element(int depth,
                                       uvm_printer_element
element);
// Main printer emit method - called by <object>.print()
extern function string emit();

endclass: json_printer


// Printer emit method - called by <object>.print():
// Takes the element at the bottom of the stack
// and starts with that:
function string json_printer::emit();
  return emit_element(0, get_bottom_element());
endfunction
```

```
// emit_element() - Note - a recursive function
//
// Uses the uvm_printer_proxy to determine if the element has
// children, if so visits them
// else prints a single element
function string json_printer::emit_element(int depth,
                                           uvm_printer_element element);
  string ret_val;
  static uvm_printer_proxy proxy = new("proxy");

  if(element != null) begin
    uvm_printer_element children[$];
    proxy.get_immediate_children(element, children);
    if(children.size() > 0) begin
      ret_val = {ret_val, open_hierarchy((depth + 1),
                                         element.get_element_name(),
                  element.get_element_type_name())};
      foreach(children[i]) begin
        // Recursive call
        ret_val = {ret_val, emit_element((depth + 1), children[i]);
      end
      ret_val = {ret_val, close_hierarchy(depth)};
    end
    else begin
      // Single level element
      ret_val = {ret_val, emit_nh_element(depth, element)};
  end
  return ret_val;
endfunction
```

The Element Stack



Overall printer element operation

// Element e0 (Parent object) pushed onto stack

// Element e1 pushed onto stack, made child of e0
// Element e1 popped off stack

// Element e2 (Array) pushed onto stack, made child of e0

// Element e3 pushed onto stack, made child of e2
// Element e3 popped off stack

// Element e4 (Child Object) pushed onto stack, made child of e0

// Element e5 pushed onto stack, made child of e4 (Child Object)
// Element e5 popped off stack

At the end of print(), the stack contains a single element representing the original object with child elements representing its fields. The parent object element is processed using the uvm_printer_element_proxy.

The JSON Printer – Part 2

```
function string json_printer::emit_element(int depth,
                                           uvm_printer_element element);
  string ret_val;
  static uvm_printer_proxy proxy = new("proxy");
  if(element != null) begin
    uvm_printer_element children[$];
    proxy.get_immediate_children(element, children);
    if(children.size() > 0) begin
      ret_val = {ret_val, open_hierarchy((depth + 1), element.get_element_name(),
                 element.get_element_type_name())};
      foreach(children[i]) begin
        // Recursive call
        ret_val = {ret_val, emit_element((depth + 1), children[i]);
      end
      ret_val = {ret_val, close_hierarchy(depth)};
    end
    else begin
      // Single level element
      ret_val = {ret_val, emit_nh_element(depth, element)};
    end
  end
  return ret_val;
endfunction
```

The JSON Printer – Part 3

```
// emit_nh_element - Print a single entry at the right depth/indent
function string json_printer::emit_nh_element(int depth,
uvm_printer_element element);
  string ret_val;
  string name;
  string value;

  name = element.get_name;
  value = element.get_value;

  ret_val = {{{depth}{indent}}};
  ret_val = {ret_val, "\"", name, "\" : \"", value, "\",\n"};
  return ret_val;

endfunction

// open_hierarchy - Start a new level in the JSON hierarchy
function string json_printer::open_hierarchy(int depth, string
element_name);
  string ret_val;

  ret_val = {{depth{ident}}};
  ret_val = {ret_val, "\"", element_name, "\" : {\n"};
  return ret_val;
endfunction

// close_hierarchy - Complete a level in the JSON hierarchy
function string json_printer::close_hierarchy(depth);
  string ret_val;

  ret_val = {{depth{indent}}};
  ret_val = {ret_val, "},\n"};
  return ret_val;
endfunction
```

Example JSON Content

```
{
  "host_txn_stream" : {
    "transaction": [
      {
        "start_time": "120ns",
        "address": "40001200",
        "write" : "1",
        "burst_length": "4",
        "write_burst": [
          "wdata[0]": "deadbeef",
          "wdata[1]": "beefdead",
          "wdata[2]": "f000baaa",
          "wdata[3]": "baaaf000"
        ],
        "write_resp": "0"
        "extension" : {
          "security": "7",
          "cache": "9",
          "coherency": "0"
        }
      },
      { ….

      },
```

```
      {
        "start_time": "156ns",
        "address": "40001200",
        "write" : "0",
        "burst_length": "4",
        "read_burst": [
          "rdata[0]": "deadbeef",
          "rresp[0]": "0",
          "rdata[1]": "beefdead",
          "rresp[1]": "0",
          "rdata[2]": "f000baaa",
          "rresp[2]": "0",
          "rdata[3]": "baaaf000"
          "rresp[3]": "0",
        ],
        "write_resp": "0"
        "extension" : {
          "security": "6",
          "cache": "8",
          "coherency": "1"
        }
      },
    ]
  }
}
```

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

THANK YOU!

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

IEEE-Compatible UVM Reference Implementation and Verification Components

Abstract Factory, Deferred Initialization, Dynamic UVM_Reg Mapping

Uwe Simm – Cadence Design Systems

UVM Factories

- UVM factory – (we known for a long time) => nothing really new
- New: UVM abstract factory
  - can now register and override abstract types (ie. **`virtual class`**)
  - Only rule: at the time of `::type_id::create()` the virtual base type needs to resolve to a concrete type via the override chain
  - Use `uvm_[object|component]_abstract[_param]_utils[_begin]` instead of `uvm_[object|component][_param]_utils[_begin]`
  - Better alignment of UVM and SV/OO

- Note: All classes in P1800.2 derived from `uvm_object` are factory enabled
  - Some constructors make this impossible (arguments w/o default values)

Abstract UVM Factory Use Case #1

```
program test01;
    import uvm_pkg::*;
`include "uvm_macros.svh"

    virtual class Animal extends uvm_object;
        `uvm_object_utils(Animal) // <- compile error
        function new(string name=""); super.new(name); endfunction
    endclass

    class Dog extends Animal;
        `uvm_object_utils(Dog)
        function new(string name=""); super.new(name); endfunction
    endclass

    initial begin
        Animal::type_id::set_type_override(Dog::get_type());
        Animal::type_id::create("somename");
    end
endprogram
```

Typical error: ‘An abstract class cannot be instantiated ..’

Abstract UVM Factory Use Case #1

```
program test02;
    import uvm_pkg::*;
`include "uvm_macros.svh"

    virtual class Animal extends uvm_object;
        `uvm_object_abstract_utils(Animal) // UVM-IEEE
        function new(string name=""); super.new(name); endfunction
    endclass

    class Dog extends Animal;
        `uvm_object_utils(Dog)
        function new(string name=""); super.new(name); endfunction
    endclass

    initial begin
        // ::create () would fail here since there is no override
        Animal::type_id::set_type_override(Dog::get_type());
        Animal::type_id::create("somename"); // factory can produce something real
    end
endprogram
```

uvm_factory Type Alias – Use Case #1

- Pre-IEEE can use only type name used during factory registration for overrides

```
class Base extends uvm_object; …  endclass

class Derived extends Base; … endclass

typedef Base myveryspecialA;

function automatic void perform();
  uvm_factory factory = uvm_factory::get();

  factory.set_type_override_by_name(”Base","Derived"); // works
  myveryspecialA::type_id::set_type_override(Derived::get_type()); // works
  factory.set_type_override_by_name("myveryspecialA","Derived"); // doesn’t work
endfunction
```

UVM Type Aliases

- In short: The type alias capability allows you to use a different string type name in factory overrides/creates other than the original type name

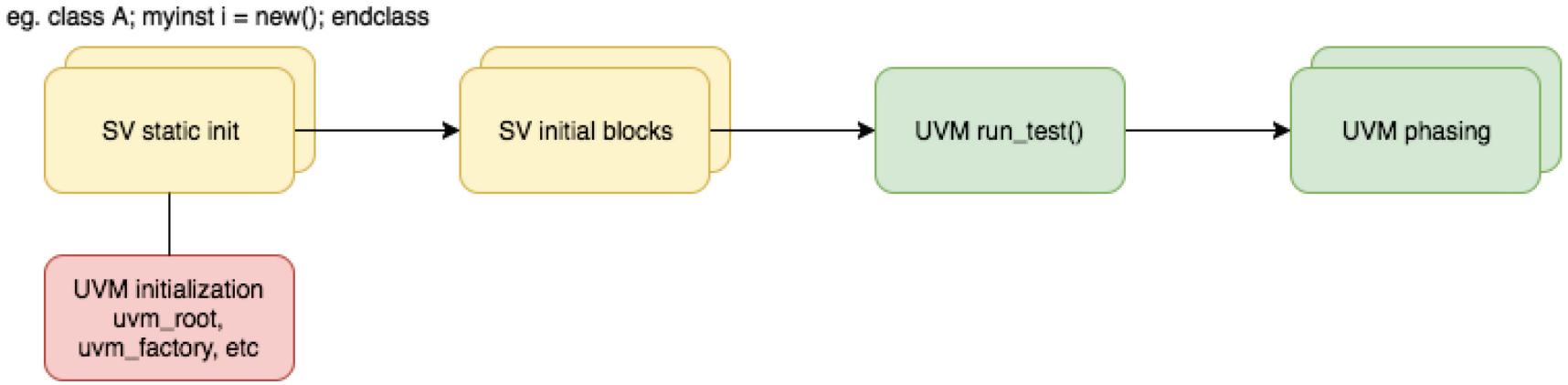
```
program test03;
    import uvm_pkg::*;
    `include "uvm_macros.svh"

    class Base extends uvm_object; … endclass

    class Derived extends Base; … endclass

    function automatic void perform();
        uvm_factory f = uvm_coreservice_t::get().get_factory();
        f.set_type_alias("myveryspecialA", Base::get_type());
        f.set_type_override_by_name("myveryspecialA",”Derived");
    endfunction
endprogram
```

UVM Type Aliases – Use Case #2

```
virtual class base extends uvm_object;
          // …
        pure virtual function void tellme();
endclass

class complex#(type T=int, T A=0)extends base;
          // …
        virtual function void tellme();
            `uvm_info("TEST",$sformatf("type T=%p A=",A),UVM_NONE)
        endfunction
endclass

string typename;
base mybase;
uvm_factory f = uvm_coreservice_t::get().get_factory();
uvm_cmdline_processor proc = uvm_cmdline_processor::get_inst();

f.set_type_alias("complex:int:5", complex#(int,5)::get_type());
f.set_type_alias("complex:string:foo", complex#(string,"foo")::get_type());
void'(proc.get_arg_value("+MY_ARG=", typename));

// pull from cmdline
     f.set_type_override_by_name("base",typename);
     mybase=base::type_id::create("myinst");
     mybase.tellme();
```

Set type alias for param type

Initialization Flow – Pre-IEEE

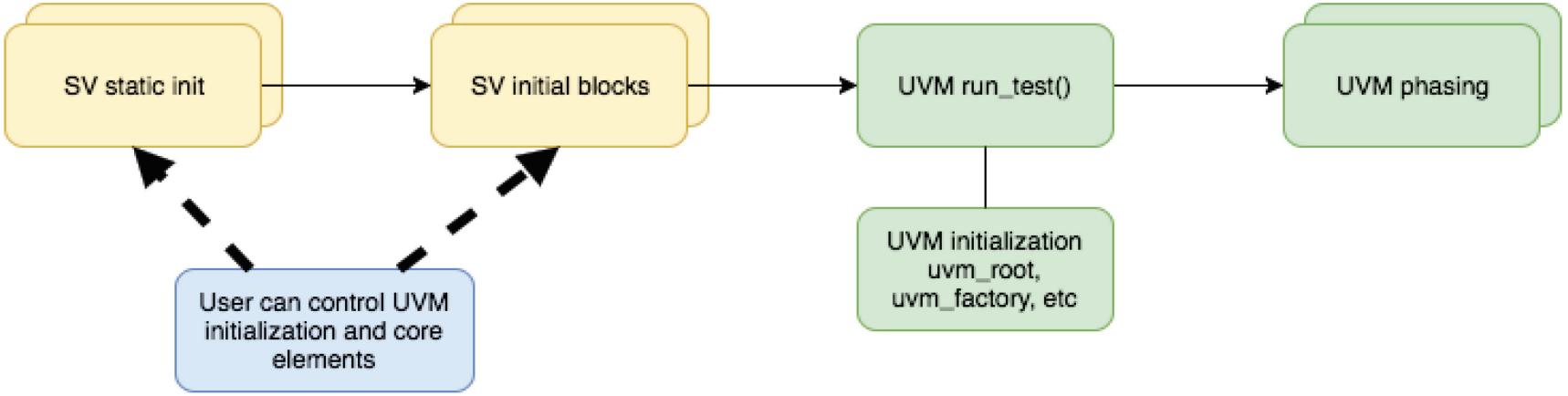


- Lots of UVM initialization happens during static initialization
- Hidden in `` `uvm_*utils ``

Why is this bad?

- No reliable order
- No guaranteed way to be the first in order to control what actual types are used for core UVM types such as factory, `uvm_root`, default report server, transaction database, policies (printer, packer, comparer, copier...)

UVM Deferred Initialization - IEEE



- Static initialization doesn't actually initialize UVM anymore
- Initialization now happens either
  - on demand if needed e.g. you want to print something, you create overrides, `uvm_coreservice_t::get()`
  - explicitly by invoking `uvm_init()`
  - at the beginning of `run_test()` if not yet performed

UVM Deferred Initialization – Default

```
program test04;
    import uvm_pkg::*;
    `include "uvm_macros.svh"

    class A extends uvm_object;
        `uvm_object_utils(A)
        function new(string name=""); super.new(name);
endfunction
    endclass

    initial begin
        // uvm_root,uvm_factory, etc not constructed yet
        $display("uvm not initialized here");

        run_test(); // <- uvm constructed here
    end
endprogram
```

No changes to standard use model

UVM Deferred Init – Use Case #1

```
program test04;
    import uvm_pkg::*;
    `include "uvm_macros.svh"

    class A extends uvm_object;
        `uvm_object_utils(A)
        function new(string name=""); super.new(name);
endfunction
    endclass

    initial begin
        // implicit construction
        uvm_config_db#(string)::set(,"scope","field","value");

        $display("uvm already initialized");

        run_test();
    end
endprogram
```

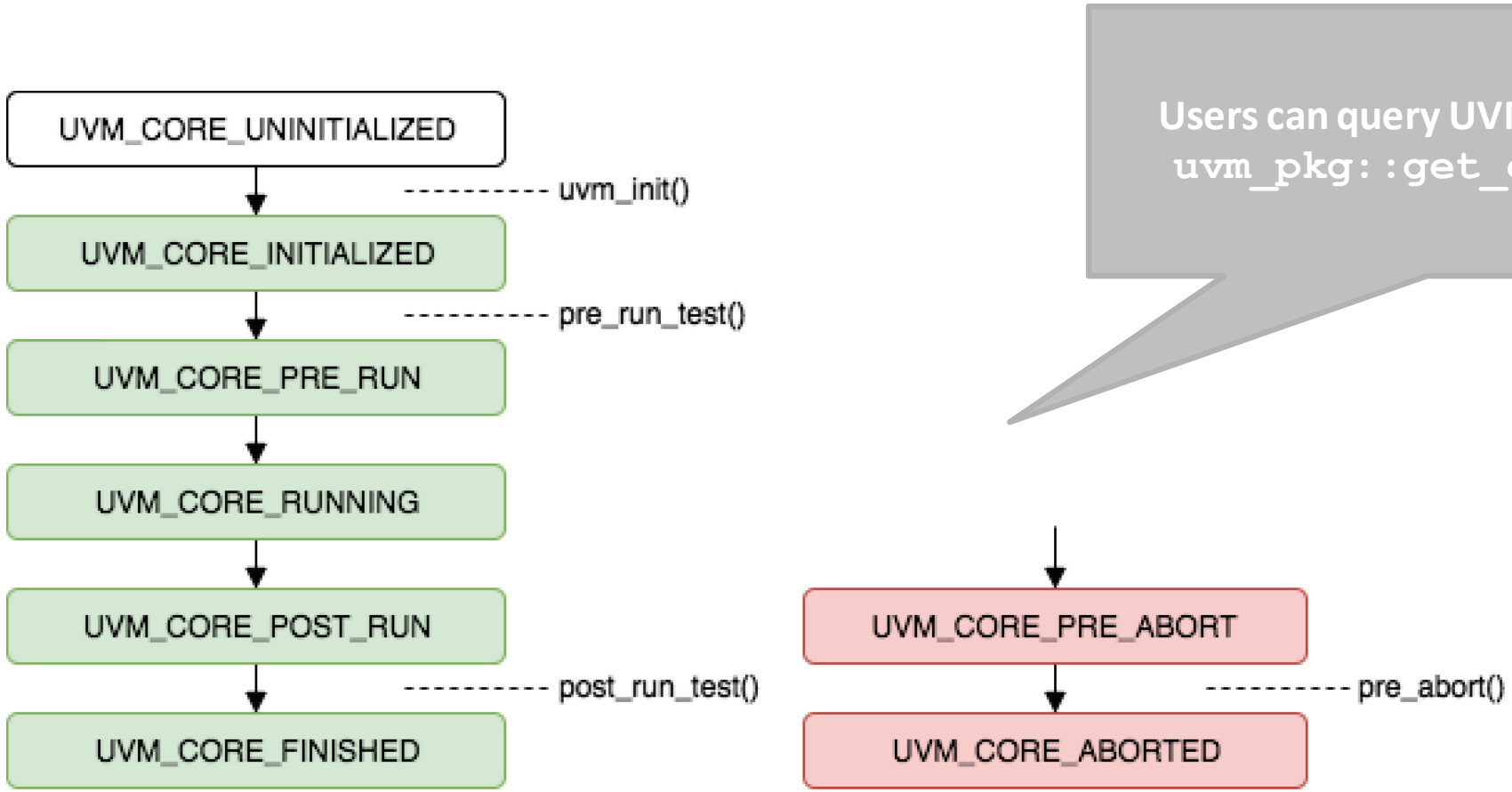
UVM initialized on-demand by actions requiring the UVM core

UVM Deferred Init – Own Factory

```
class my_coreservice_t extends uvm_default_coreservice_t;
    my_own_factory factory;
    virtual function uvm_factory get_factory();
        if(factory==null) begin
            uvm_default_factory f = new();
            factory=new;

            set_factory(factory);
        end

        return factory;
    endfunction
endclass

initial begin
    automatic my_coreservice_t cs_ = new();

    // initialize uvm with own coreservice instance
    uvm_init(cs_);
    run_test(); // test now uses my_own_factory
end
```

Create the UVM core with an own `uvm_coreservice_t` derived instance to inject an own factory type

UVM Deferred Init – Can Check Core State



Users can query UVM core state using `uvm_pkg::get_core_state()`

uvm_run_test_callback

- A callback to get a reliable notification
  - At the beginning of `run_test()` via `pre_run_test()`
  - After `run_test()` completes via `post_run_test()`
  - Before abort callbacks via `pre_abort()`
- Can be used to add actions at the beginning, end and abort of test
- Note: This is not replacement for proper run failure detection

uvm_test_callback Example

```
program test05;
    import uvm_pkg::*;
    `include "uvm_macros.svh"

    class my_run_test_callback extends uvm_run_test_callback;
        `uvm_object_utils( my_run_test_callback )

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

        function void pre_abort();
            $system($sformatf("cat xrun.log | mail -s \"test failed\" $USER"));
        endfunction
    endclass

    class test extends uvm_test; …      endclass

    initial begin
        my_run_test_callback mycb = new ("cb");
        uvm_run_test_callback::add(mycb);
        run_test();
    end
endprogram
```

Send an email with the log if simulation is aborted

UVM_REG Dynamic Address Maps #1

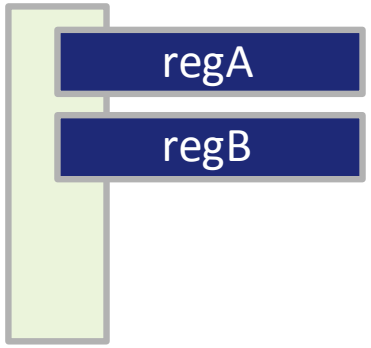
- UVM 1.2:
  - `uvm_reg_block.lock_model()` operation is to check and cache addressing
  - check is good
  - caching is for performance but not necessary for functionality
  - After `lock_model()` one cannot change topology anymore

- Problem: addressing structure is static but todays CPU subsystems/interconnect have addressing remap capabilities that cannot be modelled statically

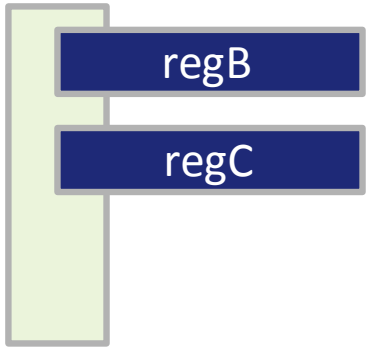
- UVM 1800.2:
  - Users can invoke `unlock_model()` and perform `lock_model()` again
  - Users can `::unregister()` elements (essentially undo the `add_*` actions)
  - Allows rebuilding the addressing hierarchy (maps) from registers as needed at runtime
    - 'kind of an interrupt" ala re-configure and continue

Example #1

mapA @0x100



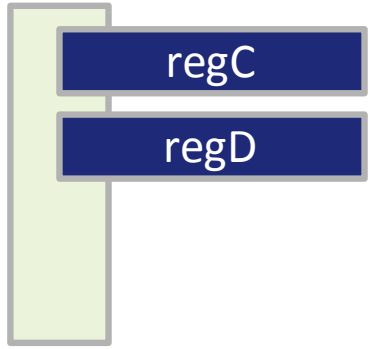
mapB @0x200



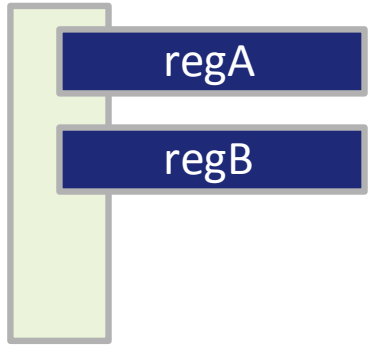
modeA

modeB

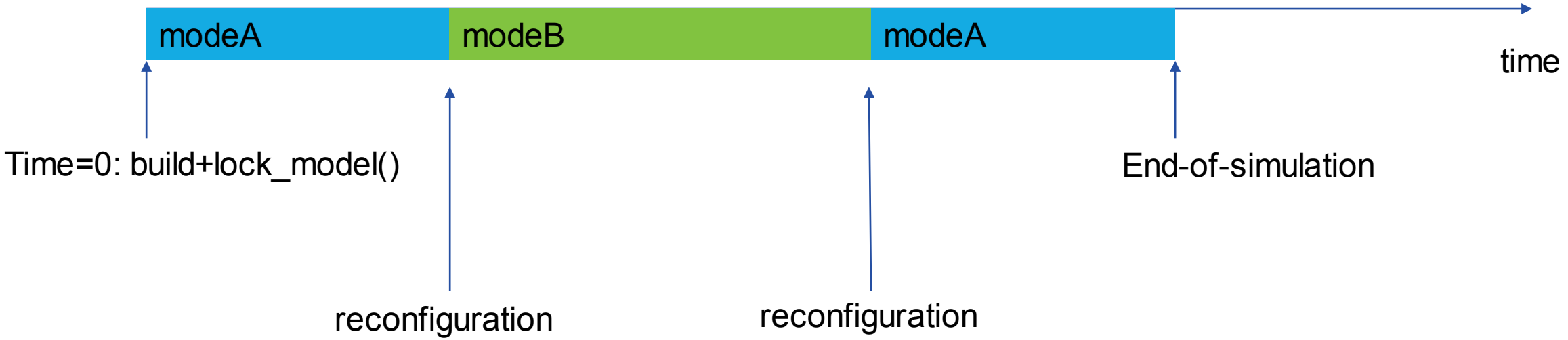
mapA @0x100



mapB @0x200



Example #1



Zero time

1. `uvm_reg_block.unlock_model()` model
2. Use (`uvm_reg_map|uvm_reg_block|uvm_reg`)`::unregister()` as needed
3. Create new maps, add registers, connect into blocks
4. Invoke `uvm_reg_block.lock_model()`

unlock Example

```
function void update_addr_map(uvm_reg_block topblk,uvm_reg_map topmap, addr_mode mode);
    topblock.unlock_model(); // undo .lock_model()

    topmap.unregister(); // decompose all relations under this map

    if(mode==modeA) begin
        topmap.mapA.add_reg(topblk.regA,'h10,"RO");
        topmap.mapA.add_reg(topblk.regA,'h14,"RO");
        topmap.mapB.add_reg(topblk.regA,'h18,"RO");
        topmap.mapB.add_reg(topblk.regA,'h1C,"RO");
        topmap.add_submap(topmap.mapA, 'h100);
        topmap.add_submap(topmap.mapB, 'h200);
    end else begin
        topmap.mapA.add_reg(topblk.regC,'h10,"RO");
        topmap.mapA.add_reg(topblk.regD,'h14,"RW");
        topmap.mapB.add_reg(topblk.regA,'h18,"RW");
        topmap.mapB.add_reg(topblk.regB,'h1C,"RO");
        topmap.add_submap(topmap.mapA, 'h100);
        topmap.add_submap(topmap.mapB, 'h200);
    end
    topblk.add_map(topmap);
    uvm_reg_block.lock_model();
endfunction
```

Unlock and unregister first

Rebuild according to current mode

Relock model

Misc. Notes for `uvm_reg`

- Changes to UVM register base classes
  - Now abstract base classes ("**`virtual`**"): `uvm_reg_field_cbs`
  - Now non-abstract classes: `uvm_reg_block`, `uvm_reg_file`, `uvm_reg`
  - `uvm_mem`: LRM=abstract but implementation will be non-abstract (LRM bug)
  - (Actually all classes matching *base are abstract now)

- `uvm_path_e` type became `uvm_door_e`
  - `UVM_DEFAULT_PATH` → `UVM_DEFAULT_DOOR`

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

THANK YOU!

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

IEEE-Compatible UVM Reference Implementation and Verification Components

Configuration, Callbacks and Reporting

Srivatsa Vasudevan – Synopsys

Build Time Control for Components

- apply_config_settings()
  - Finds all fields declared in `` `uvm_field_* `` macros
  - Applies values to those fields via a resource db lookup
  - Can be a significant contributor to build time
- UVM-1.2 :
  - Function `apply_config_settings()` is called in `uvm_component::build_phase()`
  - No control for user other than avoid `super.build_phase()`
- IEEE-1800.2 :
  - `apply_config_setttings()` is called only when (virtual) method `use_automatic_config()` returns 1

Build Time Control…

```
class default_component extends uvm_component;
  int foo;

  `uvm_component_utils_begin(default_component)
    `uvm_field_int(foo)
  `uvm_component_utils_end

  //... Constructor, etc.

  virtual function void build_phase(
                                uvm_phase phase );

    // Calls apply_config_settings to
    // configure 'foo'
    super.build_phase(phase);
  endfunction : build_phase

endclass : default_component
```

```
class explicit_component extends uvm_component;
  int foo;

  `uvm_component_utils(explicit_component)

  //... Constructor, etc.

  virtual function bit use_automatic_config();
    return 0;
  endfunction : use_automatic_config

  virtual function void build_phase(
                                     uvm_phase phase );
    // No call to apply_config_settings
    super.build_phase(phase);
    // 'foo' must be configured manually
    void(uvm_config_db#(int)::get(this, "",
                                  "foo", foo));
  endfunction : build_phase

endclass : explicit_component
```

set_local()

- Rationalisation of configuration methods that interact with resources
- Deprecated:
  - `set_int_local`
  - `set_string_local`
  - `set_object_local`
- Replaced with `set_local()` which takes a `uvm_resource_base` argument
  - A generic approach that supports all types

```
virtual function void set_local (uvm_resource_base rsrc)
```

set_local example

```
class mycomp extends uvm_component;

    typedef int int_arr[] ;
    int_arr int_arr_var = '{-1, -2, -3, -4};

    `uvm_component_utils_begin(mycomp)
      `uvm_field_array_int(int_arr_var)
    `uvm_component_utils_end

    function void set_local(uvm_resource_base rsrc);
        uvm_resource#(int_arr) rita ;
        if ($cast(rita, rsrc) &&
            (rita.get_name()=="int_arr_var"))
            int_arr_var = rita.read(this) ;
        else
            super.set_local(rsrc) ;
    endfunction

endclass
```

- You can customize which fields are in `set_local()`
- Faster…. The entire array `int_arr_var` is one call in `set_local`
- Combined `use_automatic_config()` to control `build_phase` in components

Callbacks

- All callback classes in 1800.2 now extend from `uvm_callback`
  - Including `uvm_event_callback#(T)`

- Additional introspection method added to `uvm_callbacks#(T,CB)`:

```
static function void get_all( ref CB all_callbacks[$],
                              input T obj );
```

Callback Methods

```
// Iterating over callbacks using the 1.2-era API
// still exists in 1800.2, and only sees enabled
// callbacks.
uvm_callback_iter#(my_comp_t, my_cb_t) iter;
my_cb_t cb;

iter = new(component_instance);
cb = iter.first();
while (cb != null) begin
  cb.do_something();
  cb = iter.next();
end
```

```
// Using Accellera's get_all to fetch all
// callbacks in a one shot, then checking
// for enablement.
my_cb_t cbs[$];
uvm_callbacks#(my_comp_t, my_cb_t)::get_all(cbs,
                                    component_instance);

foreach(cbs[i]) begin
  if (cbs[i].callback_mode())
    cbs[i].do_something();
end
```

Event Callbacks

```
typedef uvm_event#(int) int_evt;
typedef uvm_event_callback#(int) int_evt_cb;

class my_event_callback extends int_evt_cb;

  `uvm_object_utils(my_event_callback)

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

  virtual function void post_trigger(
                                int_evt e,
                                int data);
    //... Do something interesting here
  endfunction : post_trigger

endclass : my_event_callback
```

```
// 1.2 (Deprecated) event callbacks
initial begin
  int_evt evt = new("evt");
  my_event_callback cb = new("cb");

  evt.add_callback(cb);
  evt.delete_callback(cb);
end
```

```
// 1800.2 event callbacks
typedef uvm_callbacks#(int_evt,int_evt_cb) cbs;
initial begin
  int_evt evt = new("evt");
  my_event_callback cb = new("cb");

  cbs::add(evt, cb);
  cbs::delete(evt, cb);
end
```

Reporting – Severity/Filtering

- Errors and warnings have an associated verbosity
- In UVM1.2 library
  - Verbosity default was `UVM_LOW` for `uvm_report_error` (`UVM_NONE` for `` `uvm_error ``)
  - `uvm_report_server` would filter errors and warnings if their verbosity was above global verbosity (contradicting UVM1.2 documentation)
- In 1800.2-2017 library
  - Verbosity default is `UVM_NONE` for all errors
  - Errors and warnings are never filtered (verbosity used only if demoted to info)

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Impact of Changes

- If you used `uvm_report_error()` with default verbosity and ran with `UVM_NONE` verbosity, you may have suppressed an error which now appears.

- If you used `uvm_report_error()` with default verbosity, had a catcher demote it to info, and ran with `UVM_NONE` verbosity, you were suppressing the message before but now it appears.

- If you used `uvm_report_error()` with a specified verbosity other than `UVM_NONE`, you may have suppressed an error which now appears.

Warning Is Still Inconsistent

- IEEE 1800.2-2017 still shows
  - `uvm_report_warning` defaults to `UVM_MEDIUM`
  - `` `uvm_warning `` defaults to `UVM_NONE`
- Library matches the standard
- Because verbosity is ignored for warnings, you can see the difference only if a catcher demotes the warning to info

`uvm_do Does It All

- 1800.2 defines macro with default arguments
  - `uvm_do(SEQ_OR_ITEM, SEQR=get_sequencer(), PRIORITY=-1,CONSTRAINTS={})

- There is no extra functionality provided by
  - `uvm_do_pri(SEQ_OR_ITEM, PRIORITY)
    `uvm_do_with(SEQ_OR_ITEM, CONSTRAINTS)
    `uvm_do_pri_with(SEQ_OR_ITEM, PRIORITY, CONSTRAINTS)
    `uvm_do_on(SEQ_OR_ITEM, SEQR)
    `uvm_do_on_pri(SEQ_OR_ITEM, SEQR, PRIORITY)
    `uvm_do_on_with(SEQ_OR_ITEM, SEQR, CONSTRAINTS)
    `uvm_do_on_pri_with(SEQ_OR_ITEM, SEQR, PRIORITY, CONSTRAINTS)

Modified Deprecation

- For a cleaner API, the redundant defines should be deprecated
- BUT … A lot of existing user code likely uses these defines
- Resolution
  - The redundant defines are placed in their own file
  - That file is included in `uvm_macros.svh` only when deprecated flag is set
  - Users may compile that file after `uvm_macros.svh` if deprecated flag is not set

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

THANK YOU!

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

IEEE-Compatible UVM Reference Implementation and Verification Components

Conclusion

Justin Refice – Nvidia

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

New Library Allows

- New or existing user code to call any of the API documented in the IEEE 1800.2 LRM
- New or existing user code to call debug API carried over from UVM1.2 and documented in HTML packaged with new library
- Existing user code to call UVM1.2 API that is not in IEEE 1800.2 but does not conflict (using define `` `UVM_ENABLE_DEPRECATED_API ``)

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Why Migrate to Full 1800.2?

- All future development by Accellera will be focused on 1800.2
- If anyone develops an alternate 1800.2 library, your code will be able to use it without change
- Full 1800.2 provides more robust API, with extensibility and better consistency

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Our Request Of You

- Try the 1800.2 library with your user code
- Report issues to the Accellera Forum (http://forums.accellera.org) or ask your simulator vendor to submit an issue
- Consider participating in the UVM committee

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Future

- Near term, UVM WG will be completing the 1800.2-2017 reference implementation
  - Including resolving any issues raised by users with the Early Adopter release
- IEEE will convene to resolve errata and make some enhancements to the Register Model (and perhaps more)
- Accellera will update the library as required to match an updated IEEE specification, fix bugs, and provide potential enhancements for consideration by the IEEE.

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

THANK YOU!