IEEE 1800.2 UVM - Changes Useful UVM Tricks & Techniques

Part 1

Clifford E. Cummings

World Class Verilog, SystemVerilog & UVM Training

1639 E 1320 S, Provo, UT 84606
Voice: 801-960-1996
Email: cliffc@sunburst-design.com
Web: www.sunburst-design.com

Life is too short for bad or boring training!

Connect with Cliff on Linked in

New IEEE UVM Features

Agenda

- IEEE UVM 1800.2 Topics
  - Quick Introduction
  - Resources & References
  - Most Obvious IEEE UVM 2017 Question & Answer
  - Virtual classes & the UVM Factory
  - **`` `uvm_do ``** macro replacement
  - UVM comparators - status

Introduction

- If you were born after 1993 ← **Please raise your hand**
- UVM Best Known Methods (BKMs) … ← **Do not exist !!** ***(At least not all)***
- Frequently Asked Question: What will replace UVM? ← **In my opinion, Nothing!** ***(At least for a very long time)***

BUT … there will be modifications, simplifications and enhancements to UVM

IEEE 1800.2 is the first set of IEEE standardized enhancements to UVM

Complementary methodologies will emerge ***(such as PSS)***

PSS will help generate UVM sequences

UCIS (Unified Coverage Interoperability Standard) helps with collection of coverage data

References

You need free video registrations & two free logins

- DVCon 2018 Tutorial - IEEE-Compatible UVM Reference Implementation and Verification Components

To watch this presentation, go to:
`videos.accellera.org/videos.html`

- DVCon 2017 Tutorial - Introducing IEEE 1800.2 The Next Step for UVM

To watch this presentation, go to:
`videos.accellera.org/videos.html`

- `forums.accellera.org/`

Access the SystemVerilog and UVM Forums

Linked from
`www.accellera.org/downloads/ieee`

1800.2-2017 - IEEE UVM

- `https://ieeexplore.ieee.org/document/7932212`

Downloading PDF documents requires IEEE login
(You can create a free IEEE login account)

1800-2017 - IEEE SystemVerilog

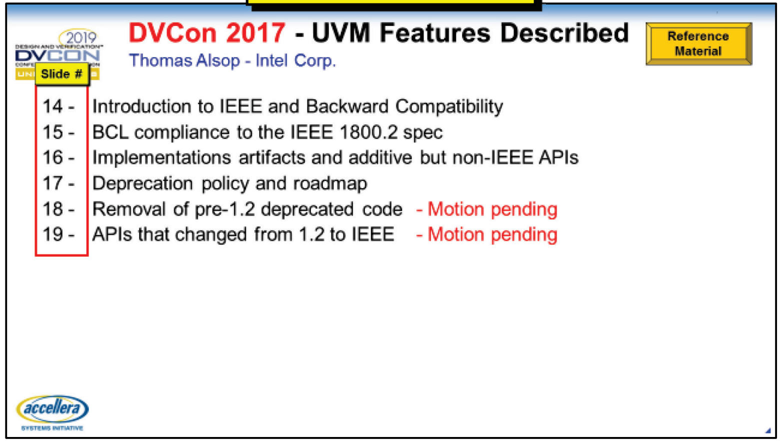
- `https://ieeexplore.ieee.org/document/8299595`

DVCon 2017 - UVM Features Described

Reference Slides at End of Presentation

Reference Material

Tom Alsop
Slides 14-19



DVCon 2017 - UVM Features Described

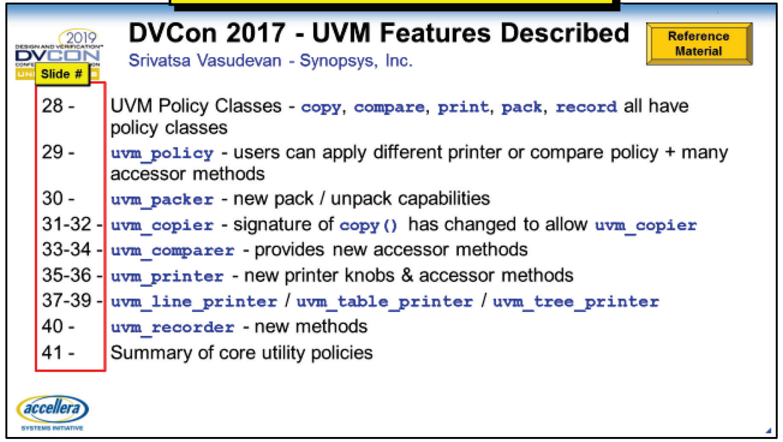
Reference Material

Thomas Alsop - Intel Corp.

Slide #

14 - Introduction to IEEE and Backward Compatibility
15 - BCL compliance to the IEEE 1800.2 spec
16 - Implementations artifacts and additive but non-IEEE APIs
17 - Deprecation policy and roadmap
18 - Removal of pre-1.2 deprecated code - Motion pending
19 - APIs that changed from 1.2 to IEEE - Motion pending

Srivatsa Vasudevan
Slides 28-41



DVCon 2017 - UVM Features Described

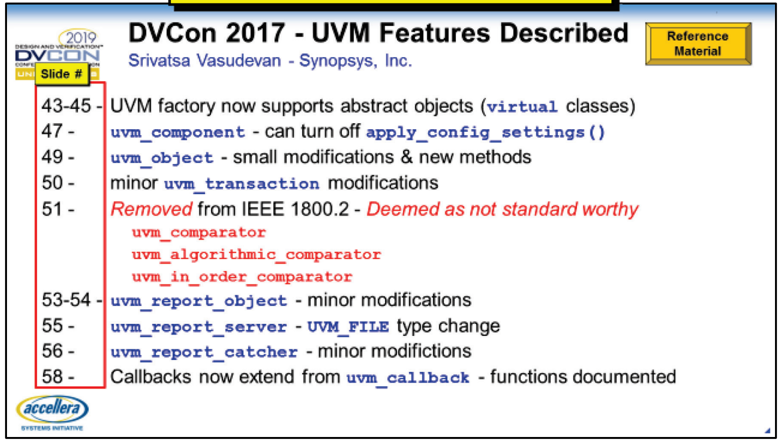
Reference Material

Srivatsa Vasudevan - Synopsys, Inc.

Slide #

28 - UVM Policy Classes - `copy`, `compare`, `print`, `pack`, `record` all have policy classes
29 - `uvm_policy` - users can apply different printer or compare policy + many accessor methods
30 - `uvm_packer` - new pack / unpack capabilities
31-32 - `uvm_copier` - signature of `copy()` has changed to allow `uvm_copier`
33-34 - `uvm_comparer` - provides new accessor methods
35-36 - `uvm_printer` - new printer knobs & accessor methods
37-39 - `uvm_line_printer` / `uvm_table_printer` / `uvm_tree_printer`
40 - `uvm_recorder` - new methods
41 - Summary of core utility policies

Srivatsa Vasudevan
Slides 43-58



DVCon 2017 - UVM Features Described

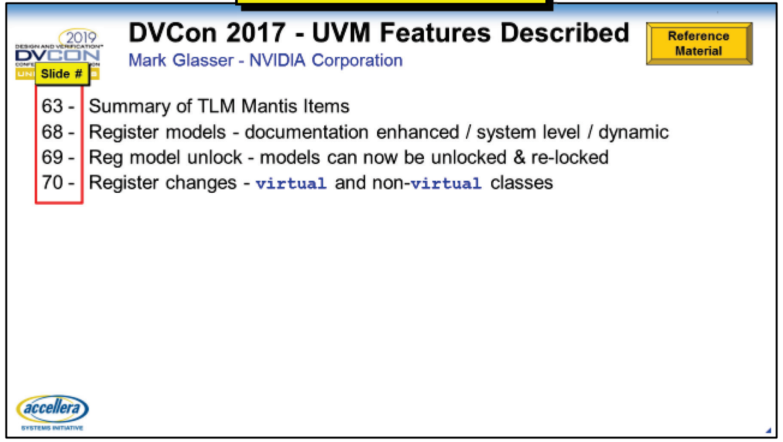
Reference Material

Srivatsa Vasudevan - Synopsys, Inc.

Slide #

43-45 - UVM factory now supports abstract objects (`virtual` classes)
47 - `uvm_component` - can turn off `apply_config_settings()`
49 - `uvm_object` - small modifications & new methods
50 - minor `uvm_transaction` modifications
51 - *Removed* from IEEE 1800.2 - *Deemed as not standard worthy*
`uvm_comparator`
`uvm_algorithmic_comparator`
`uvm_in_order_comparator`
53-54 - `uvm_report_object` - minor modifications
55 - `uvm_report_server` - `UVM_FILE` type change
56 - `uvm_report_catcher` - minor modifications
58 - Callbacks now extend from `uvm_callback` - functions documented

Mark Glasser
Slides 63-70



DVCon 2017 - UVM Features Described

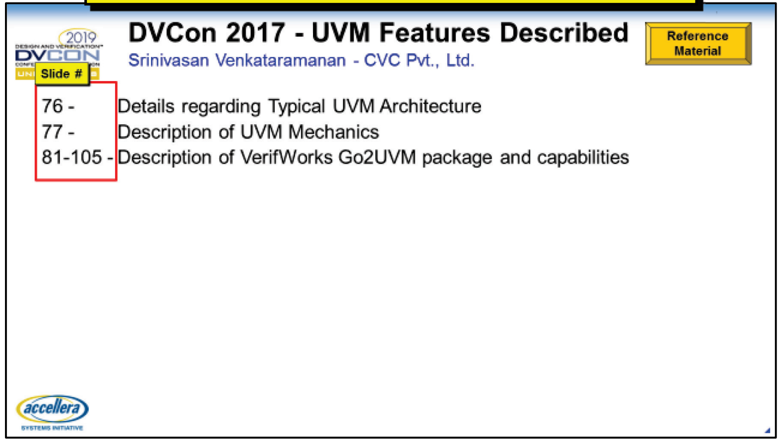
Reference Material

Mark Glasser - NVIDIA Corporation

Slide #

63 - Summary of TLM Mantis Items
68 - Register models - documentation enhanced / system level / dynamic
69 - Reg model unlock - models can now be unlocked & re-locked
70 - Register changes - `virtual` and non-`virtual` classes

Srinivasan Venkataramanan
Slides 76-105



DVCon 2017 - UVM Features Described

Reference Material

Srinivasan Venkataramanan - CVC Pvt., Ltd.

Slide #

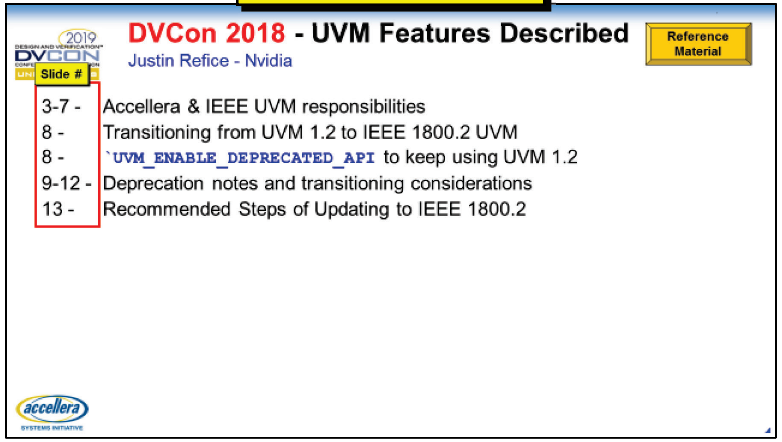
76 - Details regarding Typical UVM Architecture
77 - Description of UVM Mechanics
81-105 - Description of VerifWorks Go2UVM package and capabilities

DVCon 2018 - UVM Features Described

Reference Slides at End of Presentation

Reference Material

Justin Refice Slides 3-13



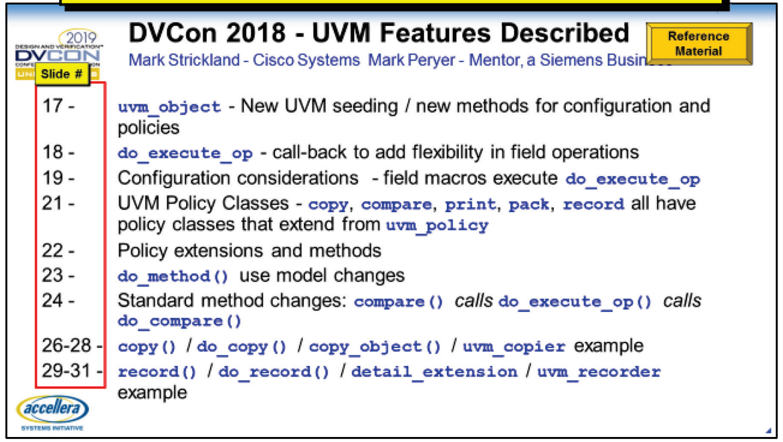
DVCon 2018 - UVM Features Described

Justin Refice - Nvidia

Reference Material

Slide #	
3-7 -	Accellera & IEEE UVM responsibilities
8 -	Transitioning from UVM 1.2 to IEEE 1800.2 UVM
8 -	`` `UVM_ENABLE_DEPRECATED_API `` to keep using UVM 1.2
9-12 -	Deprecation notes and transitioning considerations
13 -	Recommended Steps of Updating to IEEE 1800.2

Mark Strickland & Mark Peryer Slides 17-31



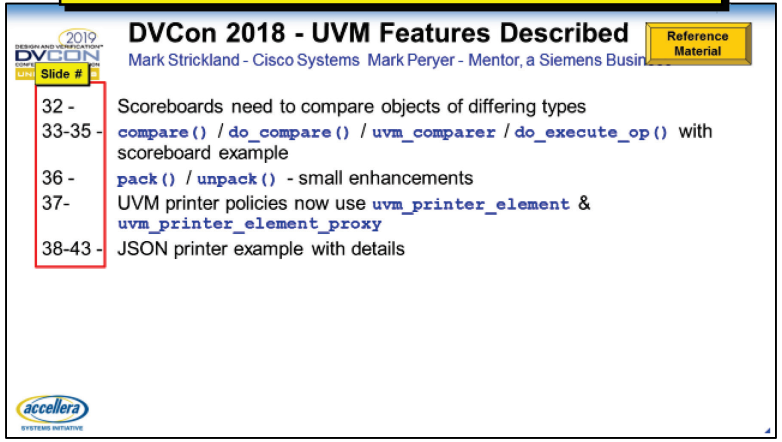
DVCon 2018 - UVM Features Described

Mark Strickland - Cisco Systems Mark Peryer - Mentor, a Siemens Busin

Reference Material

Slide #	
17 -	`uvm_object` - New UVM seeding / new methods for configuration and policies
18 -	`do_execute_op` - call-back to add flexibility in field operations
19 -	Configuration considerations - field macros execute `do_execute_op`
21 -	UVM Policy Classes - `copy`, `compare`, `print`, `pack`, `record` all have policy classes that extend from `uvm_policy`
22 -	Policy extensions and methods
23 -	`do_method()` use model changes
24 -	Standard method changes: `compare()` *calls* `do_execute_op()` *calls* `do_compare()`
26-28 -	`copy()` / `do_copy()` / `copy_object()` / `uvm_copier` example
29-31 -	`record()` / `do_record()` / `detail_extension` / `uvm_recorder` example

Mark Strickland & Mark Peryer Slides 17-31



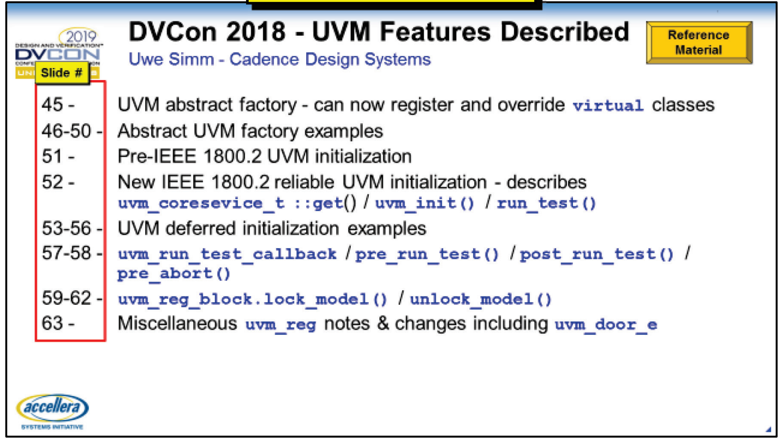
DVCon 2018 - UVM Features Described

Mark Strickland - Cisco Systems Mark Peryer - Mentor, a Siemens Busin

Reference Material

Slide #	
32 -	Scoreboards need to compare objects of differing types
33-35 -	`compare()` / `do_compare()` / `uvm_comparer` / `do_execute_op()` with scoreboard example
36 -	`pack()` / `unpack()` - small enhancements
37-	UVM printer policies now use `uvm_printer_element` & `uvm_printer_element_proxy`
38-43 -	JSON printer example with details

Uwe Simm Slides 45-63



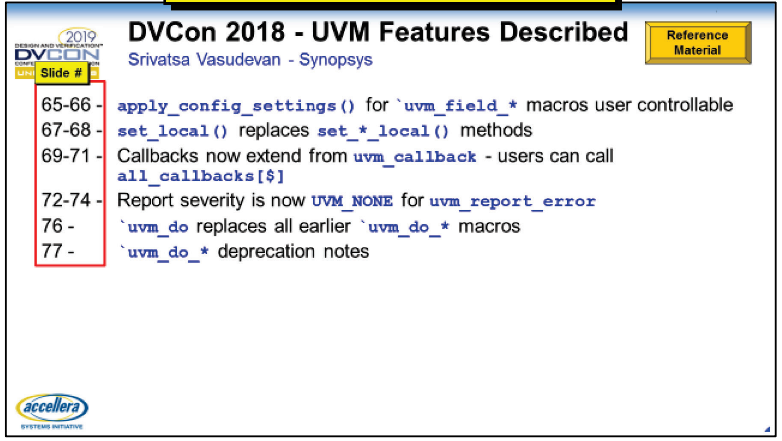
DVCon 2018 - UVM Features Described

Uwe Simm - Cadence Design Systems

Reference Material

Slide #	
45 -	UVM abstract factory - can now register and override `virtual` classes
46-50 -	Abstract UVM factory examples
51 -	Pre-IEEE 1800.2 UVM initialization
52 -	New IEEE 1800.2 reliable UVM initialization - describes `uvm_coresevice_t ::get`() / `uvm_init()` / `run_test()`
53-56 -	UVM deferred initialization examples
57-58 -	`uvm_run_test_callback` / `pre_run_test()` / `post_run_test()` / `pre_abort()`
59-62 -	`uvm_reg_block.lock_model()` / `unlock_model()`
63 -	Miscellaneous `uvm_reg` notes & changes including `uvm_door_e`

Srivatsa Vasudevan Slides 65-77



DVCon 2018 - UVM Features Described

Srivatsa Vasudevan - Synopsys

Reference Material

Slide #	
65-66 -	`apply_config_settings()` for `` `uvm_field_* `` macros user controllable
67-68 -	`set_local()` replaces `set_*_local()` methods
69-71 -	Callbacks now extend from `uvm_callback` - users can call `all_callbacks[$]`
72-74 -	Report severity is now `UVM_NONE` for `uvm_report_error`
76 -	`` `uvm_do `` replaces all earlier `` `uvm_do_* `` macros
77 -	`` `uvm_do_* `` deprecation notes

Where to Get Latest UVM BCL

Accellera Base Class Library

- Download the latest Base Class Library from Accellera web site ← **No login required**

  `http://www.accellera.org/downloads/standards/uvm`

- Latest release is: UVM 2017-1.0 Reference Implementation

Date Modified: 2018-11 ← **Just released!**

Most Obvious IEEE UVM 2017 Question

http://forums.accellera.org/

- From the UVM 2017 - Methodology and BCL Forum
- Question from Brian Hunter:

"Who can provide a summary of what is new and what has changed?"

- Response from Justin Refice

Accellera UVM Group Chair

"Wow, starting the questions off with a (*not entirely unexpected*) doozy!"

"Unfortunately *there's no single document* which states 'Here's a full list of everything that changed'. This is because a large number of changes were performed by the Accellera UVM WG prior to the IEEE UVM WG …"

Most Obvious IEEE UVM 2017 Question

http://forums.accellera.org/ - Justin Refice's Summary - Part 1

0) Removal of the User Guide

"User Guide" material removed
- It's not standard-worthy"
- DVCon 2017 - Slide 10

1) Added more `set_` / `get_` accessor methods to replace some current knobs

Knobs still work but accessor methods are a better coding practice

2) Users can insert code into the UVM core services

Advanced topic - example: create factory debugger

Allows users to make custom version of libraries without hacking existing UVM

3) Library initialization ordering

Advanced topic - but might allow "parameterized classes participating in the name-based factory"

Most Obvious IEEE UVM 2017 Question

http://forums.accellera.org/ - Justin Refice's Summary - Part 2

Justin's words

4) "Removing the Black Magic" - Field macros had undocumented behavior

Users *COULD* now implement their own field macros more safely

5) Policy class changes

All policy classes now extend `uvm_policy`

New policy class for `copy()` operations

New printer policy class extensions to implement new printers

6) Registers - "Surprisingly few changes here"

Justin's words

Most obvious change: *can now unlock and re-lock models* to remove/replace registers at runtime

Helps support hot-plugging and re-configuration designs

7) Deprecation - new methodologies / practices for handling deprecated code

Between UVM versions

Accellera DVCon Resources

http://www.accellera.org/resources/videos

Registration & viewing the videos is FREE!

***Justin* - "At DVCon 2017 & 2018, there were tutorials which covered all of the above and more, with detailed examples."**

- U.S. DVCon 2018 Presentation by:
  - Justin Refice - *Nvidia*
  - Mark Strickland - *Cisco Systems*
  - Mark Peryer - *Mentor, a Siemens Business*
  - Uwe Simm - *Cadence Design Systems*
  - Srivatsa Vasudevan - *Synopsys*
- U.S. DVCon 2017 Presentation by:
  - Thomas Alsop - *Intel*
  - Srivatsa Vasudevan - *Synopsys*
  - Mark Glasser - *Nvidia*
  - Srinivasan Venkataramanan - *CVC Pvt., Ltd.*
  - Krishna Thottempudi - Qualcomm

#1 Added more `set_` / `get_` accessor methods to replace some current knobs

***Justin* - "*Aside from #1*, most of those changes are for *advanced use cases*, or providers of infrastructure. *Day-to-day users shouldn't necessarily see a drastic change.*"**

DVCon 2017 & 2018 Tutorials

- Multiple features shared but most were very complex corner-case enhancements

(Complex) examples in the DVCon presentation slides

- Personally, I never tried to implement the corner-case functionality:

Many examples were very difficult to understand

Except to the presenter!

I personally barely followed the complex examples

- I could re-show:
  - Excellent examples from DVCon presentations
  - And show advanced corner-case topics that most would barely understand

I am not going to do that

I want to show you more mainstream enhancement examples

DVCon 2017 & 2018 Tutorials

- Justin's list of 1800.2 features shows topics covered in the DVCon presentations

Register for *free access* on videos.accellera.org

- Doing anything tricky or complex?

Please *review the excellent examples* that you will find in the DVCon presentations

- See the slides and hear the explanations by the actual presenters

***Presentation audio always includes more* than the presentation slides**

If you are *doing anything complex*, it is worth a listen

New UVM Features Will Be Shown

- This is Cliff's way of saying these guys are really smart!

… and Cliff is really average!

- *To Be Shown:* Enhancement features that the average UVM coder can use

- Where appropriate: List DVCon slides where you can find more info

- I will also show you a few of my favorite tricks

To make your attendance worth while

Virtual Classes

Purpose and Usage

- `virtual` classes - only intended to be a base class

Not enough functionality to use as stand-alone constructed objects

Most UVM components must be extended to be useful - so they are `virtual` classes

- `virtual` class methods can be `virtual` or non-`virtual`
  - non-`virtual` methods means extended class can override and change the prototype

Prototype = function/task header

Polymorphism not possible with non-virtual methods

  - `virtual` methods create placeholders with required prototype

Same function/task header

Can include default implementation if the extended class does not override the method.

Virtual Classes

Purpose and Usage

- You want `virtual` classes to have `virtual` methods

`virtual` methods make upcasting and polymorphism possible

- SystemVerilog-2009 added `pure virtual` methods

Just like virtual methods -
Requires the same prototype

Unlike virtual methods -
There can be no default method implementation

- `pure virtual` methods REQUIRE extended classes to override the method

Extended class *must* provide an implementation

`pure` keyword is only legal in a `virtual` class

Pure Virtual Methods

Two important purposes

```
virtual class vc1a;
  bit [7:0] a;

  pure virtual function void seta(bit [7:0] val);
endclass
```

pure virtual method

(1) pure virtual methods can only be a method prototype

No method body allowed

No endfunction / endtask allowed

```
class ex1a extends vc1a;

  virtual function void seta(bit [7:0] val);
    a = val;
  endfunction
endclass
```

ex1a *MUST* override seta()
(must provide an implementation)

(2) pure virtual methods ***must*** be overridden in a non-virtual class

NOTE: pure keyword is only legal in virtual classes

Pure Virtual Methods

virtual classes

```
virtual class vc1a;
  bit [7:0] a;

  pure virtual function void seta(bit [7:0] val);
endclass
```

pure virtual method

```
virtual class vc1b;
  bit [7:0] a;

  pure virtual function void seta(bit [7:0] val);
endclass
```

pure virtual method

```
virtual class vc2a extends vc1a;


endclass
```

vc2a *does NOT* override seta() method

```
virtual class vc2b extends vc1b;

   virtual function void seta(bit [7:0] val);
    a = val;
  endfunction
endclass
```

vc2b *DOES* override seta() method

non-virtual classes

```
class ex1a extends vc2a;

  virtual function void seta(bit [7:0] val);
    a = val;
  endfunction
endclass
```

ex1a *MUST* override seta()
(must provide an implementation)

```
class ex1b extends vc2b;

  // optional override of seta()
endclass
```

ex1b can *OPTIONALLY* override seta()

Prior to Pure Virtual?

How was the pure-virtual functionality implemented?

- Engineers would code virtual methods with a simple implementation

To display a *fatal* message that the method had not been overridden

VMM had some of these non-pure virtual methods

```
virtual class uvm_subscriber ...
                     extends uvm_component;
  ...
  virtual function void write(T t);
    `uvm_fatal("ERR", "Must implement write()")
  endfunction
endclass
```

***UVM-like* non-pure virtual method with Fatal message**

- Comparing `virtual` -vs- `pure virtual`:
  - `virtual` methods ← **Missing implementations were discovered at *run*-time** — **Very late to discover the missing implementation**
  - `pure virtual` methods ← **Missing implementations are discovered at *compile*-time** — **Problems are found sooner and resolved quicker**

Two Common Testbench Base Classes

Common User-Defined Base Classes

- User-defined classes that should not be directly created:
  - `test_base`

    Common test functionality

```
class test_base extends uvm_test;       ...
class test1 extends test_base;          ...
```

  - `vseq_base`

    Declares subsequecer handles and retrieves / checks the handles from the virtual sequencer

```
class vseq_base extends uvm_sequence;   ...
class vseq1 extends vseq_base;          ...
```

- In UVM, these cannot be `virtual` classes

Virtual classes cannot be factory-created

UVM compilation errors if put in the factory

Typical error: "An abstract class cannot be instantiated .."

Virtual Classes in the Factory

UVM 1800.2 Enhancement - For uvm_objects

- Utils-macros for Classes:

```
`define uvm_object_utils(T)

`define uvm_object_utils_begin(T)
`define uvm_object_utils_end

`define uvm_object_param_utils(T)

`define uvm_object_param_utils_begin(T)
`define uvm_object_param_utils_end
```

- Utils-macros for Virtual Classes:

```
`define uvm_object_abstract_utils(T)

`define uvm_object_abstract_utils_begin(T)
`define uvm_object_abstract_utils_end

`define uvm_object_abstract_param_utils(T)

`define uvm_object_abstract_param_utils_begin(T)
`define uvm_object_abstract_utils_end
```

Now `virtual` base classes for transactions and sequences can be stored in the factory

NOTE: Now you can store `virtual` classes with `pure virtual` methods in the factory

Virtual Classes in the Factory

UVM 1800.2 Enhancement - For uvm_components

- Utils-macros for Classes:

```
`define uvm_component_utils(T)

`define uvm_component_utils_begin(T)
`define uvm_component_utils_end

`define uvm_component_param_utils(T)

`define uvm_component_param_utils_begin(T)
`define uvm_component_param_utils_end
```

- Utils-macros for Virtual Classes:

```
`define uvm_component_abstract_utils(T)

`define uvm_component_abstract_utils_begin(T)
`define uvm_component_abstract_utils_end

`define uvm_component_abstract_param_utils(T)

`define uvm_component_abstract_param_utils_begin(T)
`define uvm_component_abstract_utils_end
```

Now `virtual` base classes for tests and other components can be stored in the factory

NOTE: Many of the UVM virtual base classes are now factory enabled using the abstract_utils macros

Testbench & Factory Access

UVM 1.1d

- UVM 1.1d allowed access to the **factory** handle

```
class test_base extends uvm_test;
  ...

  function void start_of_simulation_phase(uvm_phase phase);
    super.start_of_simulation_phase(phase);
    this.print();
    factory.print();
  endfunction

  ...
endclass
```

start_of_simulation phase
(after the testbench is built and connected)

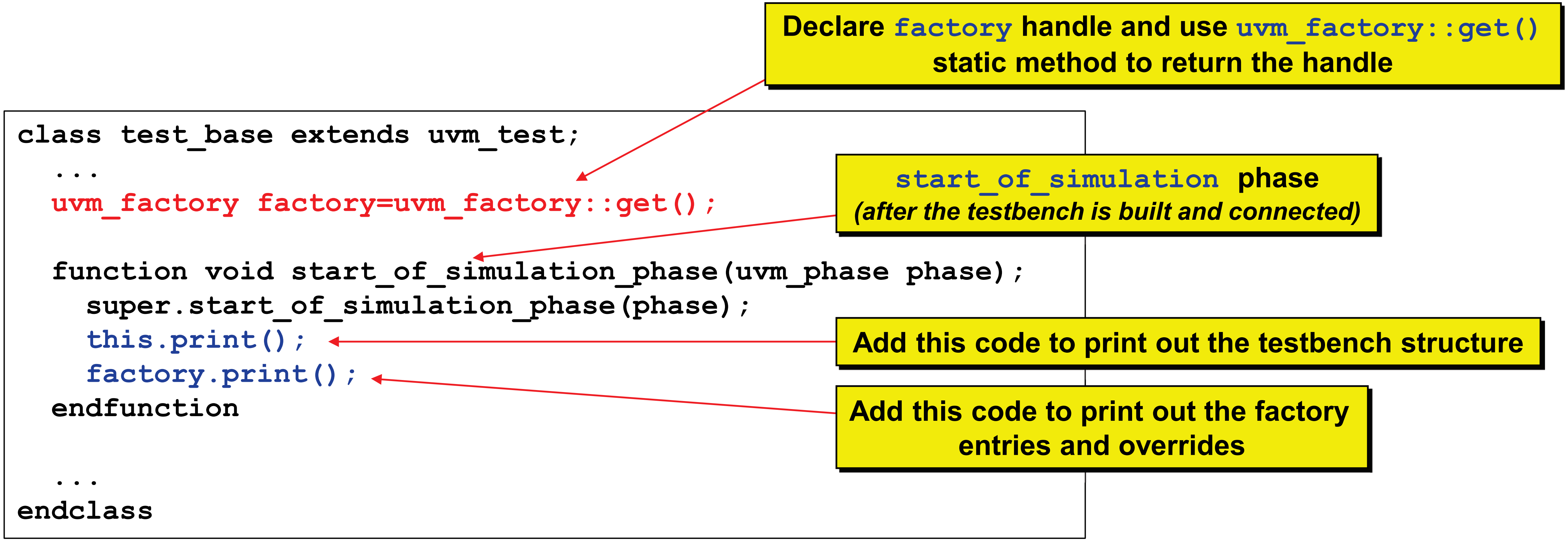
Add this code to print out the testbench structure

Add this code to print out the factory entries and overrides

Testbench & Factory Access

UVM 1.2 & 1800.2

- UVM 1.2 & 1800.2 require declaration of the `factory` handle



```
class test_base extends uvm_test;
  ...
  uvm_factory factory=uvm_factory::get();

  function void start_of_simulation_phase(uvm_phase phase);
    super.start_of_simulation_phase(phase);
    this.print();
    factory.print();
  endfunction

  ...
endclass
```

Declare `factory` handle and use `uvm_factory::get()` static method to return the handle

`start_of_simulation` phase *(after the testbench is built and connected)*

Add this code to print out the testbench structure

Add this code to print out the factory entries and overrides

`uvm_do Macros

UVM 1.2 -vs- UVM 1800.2

`uvm_do sequence or sequence item

`uvm_do actions

		Macro Inputs				UVM actions			
		SEQ_OR_ITEM	SEQUENCER	PRIORITY	{CONSTRAINTS}	create()	start_item()	randomize()	finish_item()
✓ Common	`uvm_do(I)	X				X	X	X	X
✘	`uvm_do_with(I,{C}) Deprecated	X			X	X	X	X	X
✘ Common vsequencer	`uvm_do_on(I,S) Deprecated	X	X			X	X	X	X
✘	`uvm_do_on_with(I,S,{C}) Deprecated	X	X		X	X	X	X	X
✘	`uvm_do_pri(I,P) Deprecated	X		X		X	X	X	X
✘ Less Common	`uvm_do_pri_with(I,P,{C}) Deprecated	X		X	X	X	X	X	X
✘	`uvm_do_on_pri(I,S,P) Deprecated	X	X	X		X	X	X	X
✘	`uvm_do_on_pri_with(I,S,P,{C}) Deprecated	X	X	X	X	X	X	X	X

`uvm_create, `uvm_send, `uvm_rand Macros

UVM 1.2 -vs- UVM 1800.2

`uvm_macro sequence or sequence item

`uvm_macro actions

Less frequently used macros

		Macro Inputs				UVM actions			
		SEQ_OR_ITEM	SEQUENCER	PRIORITY	{CONSTRAINTS}	create()	start_item()	randomize()	finish_item()
✓	`uvm_create(I)	X				X			
✗	~~`uvm_create_on(I,S)~~ Deprecated	X	X			X			
✓	`uvm_send(I)	X					X		X
✗	~~`uvm_send_pri(I,P)~~ Deprecated	X		X			X		X
✓	`uvm_rand_send(I)	X					X	X	X
✗	~~`uvm_rand_send_with(I,{C})~~ Deprecated	X			X		X	X	X
✗	~~`uvm_rand_send_pri(I,P)~~ Deprecated	X		X			X	X	X
✗	~~`uvm_rand_send_pri_with(I,P,{C})~~ Deprecated	X		X	X		X	X	X

New 1800.2 `uvm_do Commands

`uvm_do `uvm_create `uvm_send `uvm_rand_send

```
`uvm_do(SEQ_OR_ITEM, SEQR=get_sequencer(), PRIORITY=-1, CONSTRAINTS={})
```

UVM 1.2 usage examples

```
`uvm_do_on(ahb_seq, ahb_sqr)
`uvm_do_on(eth_seq, eth_sqr)
```

```
`uvm_do_with(tr, {rw_type==WRITE;})
```

UVM 1800.2 usage examples

```
`uvm_do(ahb_seq, ahb_sqr)
`uvm_do(eth_seq, eth_sqr)
```

Virtual sequences

```
`uvm_do(tr, , , {rw_type==WRITE;})
```

Transaction w/constraint

... AND before you ask !

```
`uvm_do(.SEQ_OR_ITEM(tr), .CONSTRAINTS({rw_type==WRITE;}))
```

NO ... you cannot pass values by argument name

```
    `uvm_create(SEQ_OR_ITEM, SEQR=get_sequencer())
      `uvm_send(SEQ_OR_ITEM, PRIORITY=-1)
 `uvm_rand_send(SEQ_OR_ITEM, PRIORITY=-1, CONSTRAINTS={})
```

UVM Comparator Classes

DVCon 2017 - Slide 51

- Removed from P1800.2

```
uvm_comparator
uvm_algorithmic comparator
uvm_in_order_comparator
```

Deemed as not standard-worthy"
DVCon 2017 - Slide 51

NOT Deprecated:
The Source files are still there

These are some of my favorite
UVM 1800.2 new features

Some of Cliff's favorite UVM topics

- Cliff's favorite UVM topics
  - UVM transaction - why is it a class?
  - UVM `do_`methods -vs- field macros
  - `start_item()` / `finish_item()` -vs- `` `uvm_do ``
  - UVM messaging macros, tricks & guidelines
  - UVM factory & `factory.print()`
  - Analysis paths

Why Is UVM Hard To Learn?

- UVM User Guide was written by Cadence

Teaches *Cadence* recommended methods

Uses a large number of UVM macros

- UVM tutorials by Mentor on
  `VerificationAcademy.org`

Teaches *Mentor* recommended methods

Fewer UVM macros / more UVM method calls

- OVM Cookbook written by Mentor employees

Based on earlier versions of OVM

- User Guide, tutorials and Cookbook do not acknowledge alternate methods

Users think one or more sources have bugs

- Authors of UVM materials are really, really smart software engineers

Authors assume everyone knows SV, OO and polymorphism

Authors don't know how to teach the concepts to beginners

UVM Transaction Base Classes

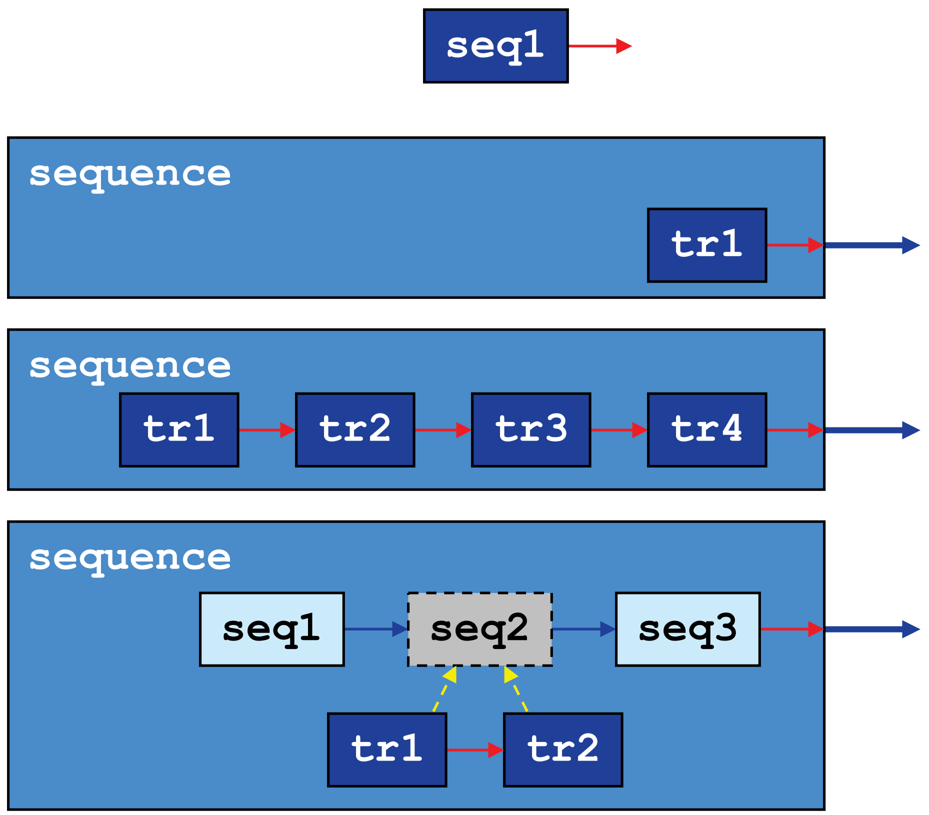
Good reference paper:
UVM Transactions - Definitions, Methods and Usage
`www.sunburst-design.com/papers/CummingsSNUG2014SV_UVM_Transactions.pdf`

Transactions & Sequences

What Is Their Composition?

Basic transactions are extended from `uvm_sequence_item`

- Transactions are driven into the dut_if
- Sequences can be built from:
  - A single transaction
  - Multiple transactions
  - Multiple other sequences



Transaction Data

Why use classes? Why not use structs?

- Classes - dynamic

✓ Multiple fields

✓ rand fields

✓ Randomization constraints

✓ Built-in methods

✓ Generate as many as needed at run time

✓ Classes can be extended

Allows more than one transaction type with a common base type

✓ Can be in a factory for run-time substitution

Classes are basically dynamic, ultra flexible structs that can
- **be easily randomized**
- **easily control the randomization**
- **be created whenever they are needed**

- Structs - static

✓ Multiple fields

✗ NO rand fields

✗ NO randomization constraints

✗ NO built-in methods

✗ Must anticipate & statically declare all structs at the beginning of the simulation

✗ Structs must be copied

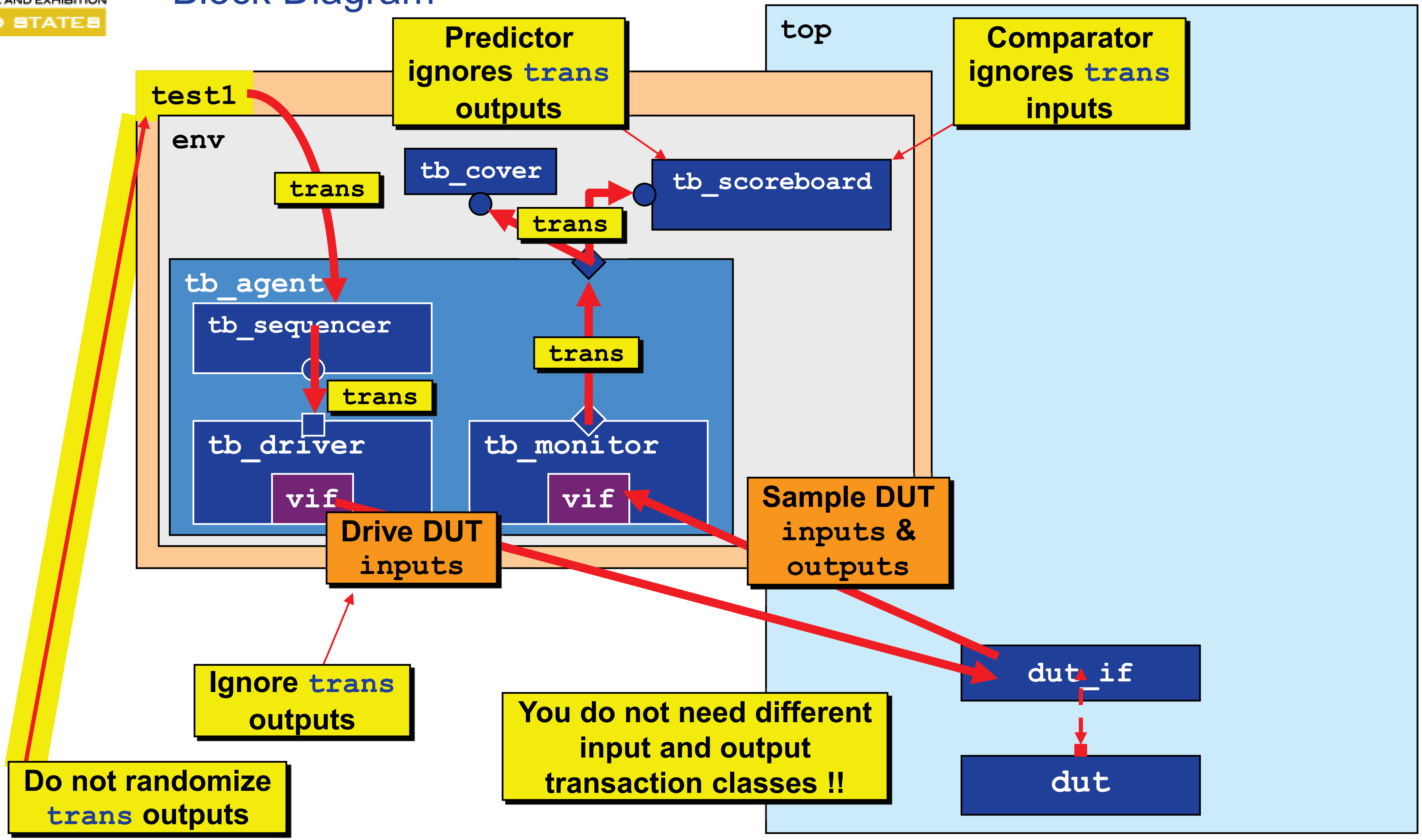
Copies are modified if more than one transaction type is desired

✗ No factories for structs

The default transaction type used by UVM components is `uvm_sequence_item`

Passing Transactions & Signals

Block Diagram



Predictor ignores trans outputs

Comparator ignores trans inputs

Drive DUT inputs

Sample DUT inputs & outputs

Ignore trans outputs

You do not need different input and output transaction classes !!

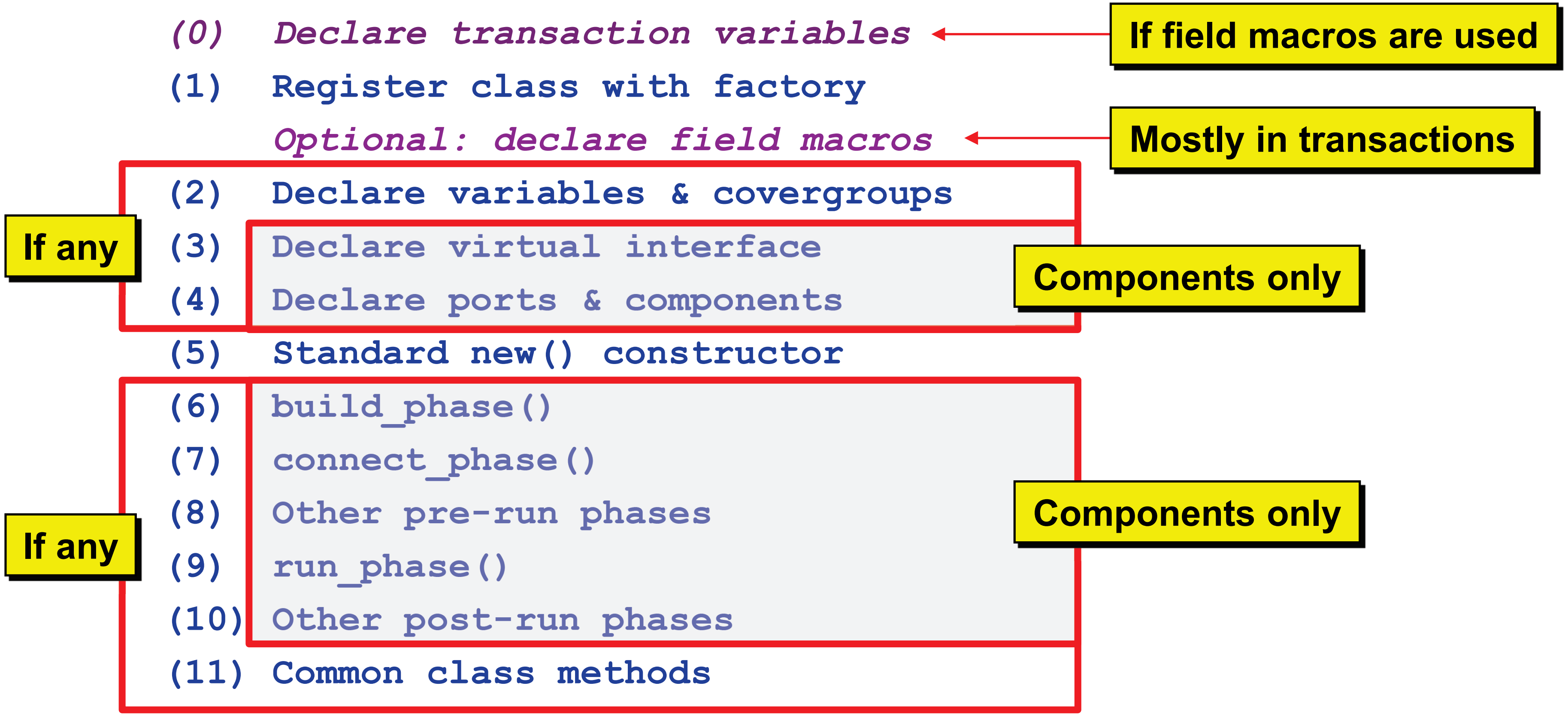
Do not randomize trans outputs

Standardized UVM Formatting

Standard UVM Coding Style

Cliff's preferred styles

- UVM testbench components and UVM transaction definitions



(0) Declare transaction variables ← If field macros are used

(1) Register class with factory

Optional: declare field macros ← Mostly in transactions

(2) Declare variables & covergroups

(3) Declare virtual interface

(4) Declare ports & components

(5) Standard new() constructor

(6) build_phase()

(7) connect_phase()

(8) Other pre-run phases

(9) run_phase()

(10) Other post-run phases

(11) Common class methods

If any

Components only

If any

Components only

UVM Transactions Styles

do_methods() -vs- field macros

- Using **do**_*methods*()

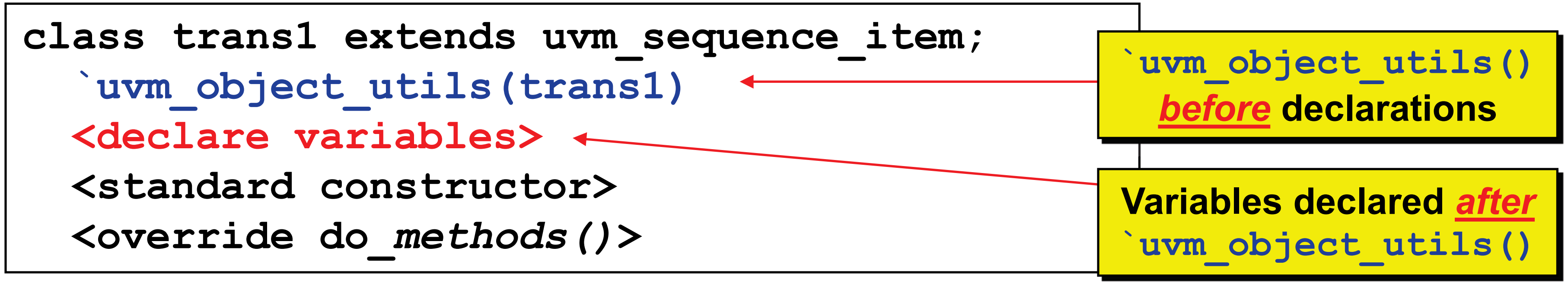
```
(1) Register with factory

(2) Declare vars/~~covergroups~~
(5) new() constructor
(11) Common trans methods
       convert2string()
       do_copy() / do_compare()
       other do_methods()
```

- Using field macros

```
(0) Declare trans vars
(1) Register with factory
    Optional: field macros
(2) Declare vars/~~covergroups~~
(5) new() constructor
(11) Common trans methods
       convert2string()
```

`uvm_object_utils Macro Usage

- Using do_*methods*()

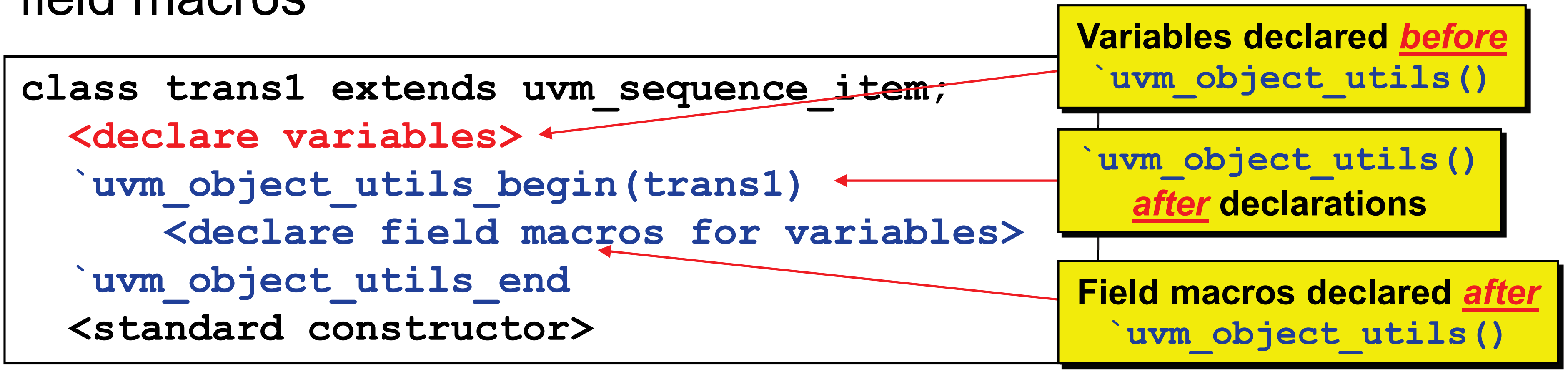


```
class trans1 extends uvm_sequence_item;
  `uvm_object_utils(trans1)
  <declare variables>
  <standard constructor>
  <override do_methods()>
```

`uvm_object_utils() ***before*** declarations

Variables declared ***after*** `uvm_object_utils()

- Using field macros



```
class trans1 extends uvm_sequence_item;
  <declare variables>
  `uvm_object_utils_begin(trans1)
      <declare field macros for variables>
  `uvm_object_utils_end
  <standard constructor>
```

Variables declared ***before*** `uvm_object_utils()

`uvm_object_utils() ***after*** declarations

Field macros declared ***after*** `uvm_object_utils()

Standard Transaction Methods

Standard Transaction Methods

Defined in `uvm_object()` base class

- 11 Standard Transaction Methods

```
  copy(),
compare(),
 print(),        sprint(),
  pack(),    pack_bytes(),    pack_ints(),
unpack(), unpack_bytes(), unpack_ints(),
record()
```

`copy()` & `compare()` are *very important*

Somewhat important

Used for serial-to-parallel applications

For debugging transactions

- 3 more transaction methods

```
create(),
clone(),
convert2string()
```

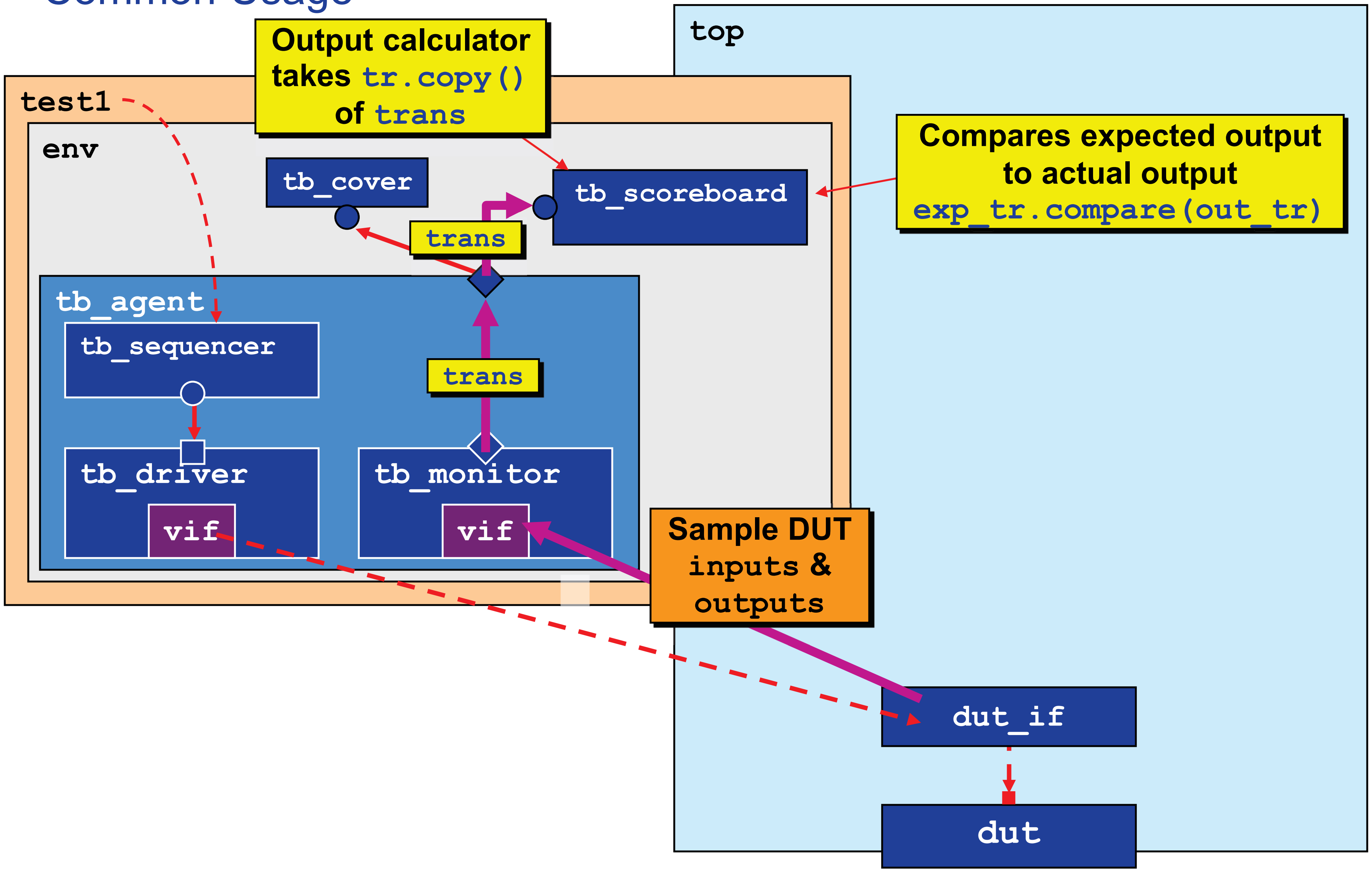
Auto-generated by `` `uvm_object_utils() `` macro

Creates and copies a transaction

`convert2string()` is *very important*

copy() & compare() Usage

Common Usage



Output calculator takes tr.copy() of trans

Compares expected output to actual output exp_tr.compare(out_tr)

Sample DUT inputs & outputs

Implementing Transaction Methods

For User-Defined sequence_items

- Each transaction should include important methods

- Two ways to implement important transaction methods:
  - Field macros
  - Manual coding

Simple - but *inefficient* (*simulations*)

These are shown in the UVM User Guide

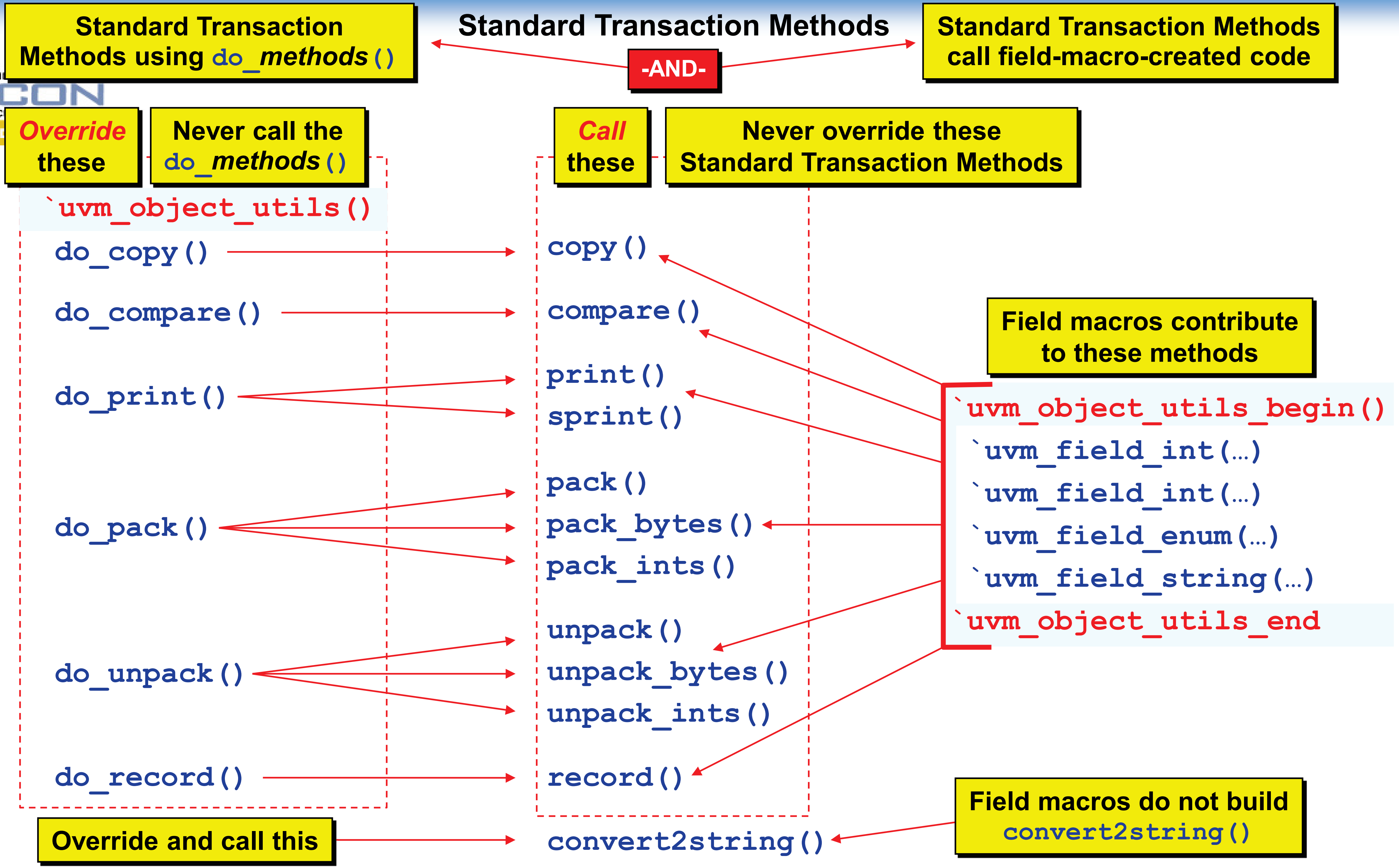
Cadence recommends using these macros

Mentor recommends avoiding these macros

Not too difficult - *more efficient* (*simulations*)

These are shown on Verification Academy

Mentor recommends coding the methods
(using user-defined hooks - next slides)



Standard Transaction Methods

-AND-

Standard Transaction Methods using do_*methods*()

Standard Transaction Methods call field-macro-created code

Override these

Never call the do_*methods*()

Call these

Never override these Standard Transaction Methods

```
`uvm_object_utils()
  do_copy()
  do_compare()
  do_print()
  do_pack()
  do_unpack()
  do_record()
```

```
copy()
compare()
print()
sprint()
pack()
pack_bytes()
pack_ints()
unpack()
unpack_bytes()
unpack_ints()
record()
```

Field macros contribute to these methods

```
`uvm_object_utils_begin()
  `uvm_field_int(…)
  `uvm_field_int(…)
  `uvm_field_enum(…)
  `uvm_field_string(…)
`uvm_object_utils_end
```

Override and call this

```
convert2string()
```

Field macros do not build convert2string()

Why Not Override compare() Method?

– Insert your compare code on line 58 or line 59

Are you kidding me ??

```
function bit uvm_object::compare (uvm_object rhs, ...
```

57 lines of pre-compare() code

Line 58 - Call the field-macros compare() code

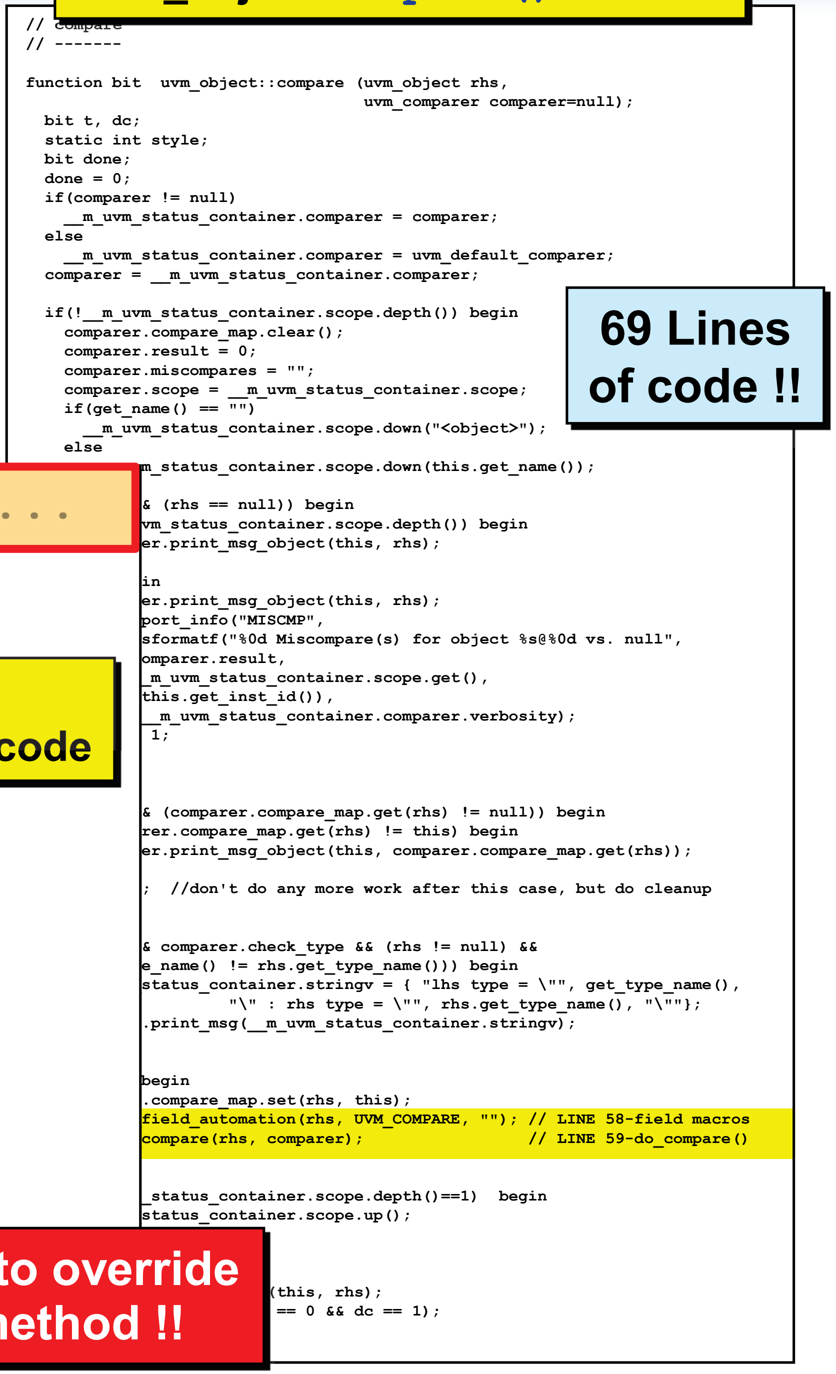
```
__m_uvm_field_automation(rhs, UVM_COMPARE, "");
dc = do_compare(rhs, comparer);
```

Line 59 - Call the do_compare() code

10 lines of post-compare() code

```
endfunction
```

uvm_object compare() method



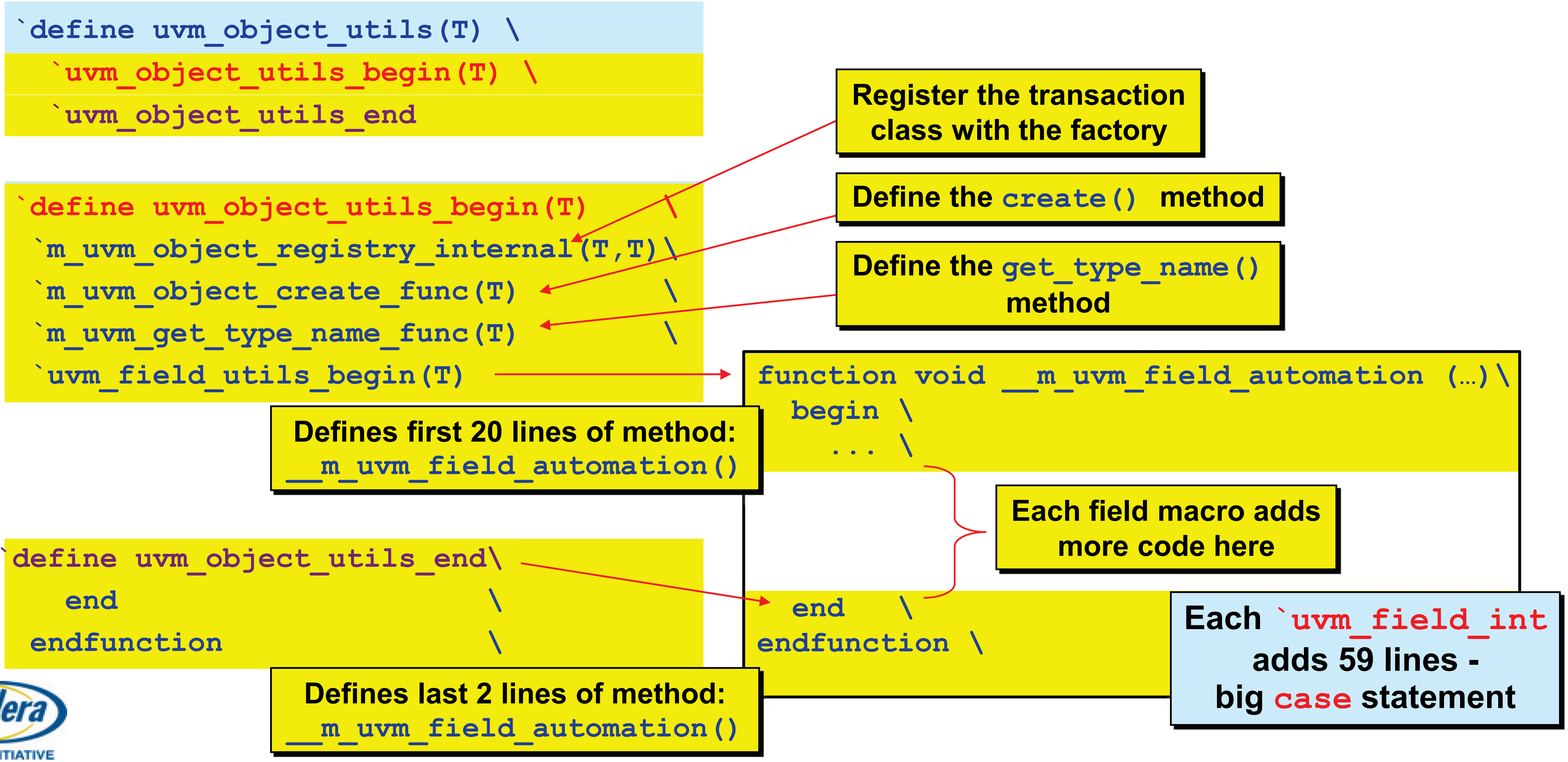
```
// compare
// -------

function bit  uvm_object::compare (uvm_object rhs,
                                   uvm_comparer comparer=null);
  bit t, dc;
  static int style;
  bit done;
  done = 0;
  if(comparer != null)
    __m_uvm_status_container.comparer = comparer;
  else
    __m_uvm_status_container.comparer = uvm_default_comparer;
  comparer = __m_uvm_status_container.comparer;

  if(!__m_uvm_status_container.scope.depth()) begin
    comparer.compare_map.clear();
    comparer.result = 0;
    comparer.miscompares = "";
    comparer.scope = __m_uvm_status_container.scope;
    if(get_name() == "")
      __m_uvm_status_container.scope.down("<object>");
    else
             m_status_container.scope.down(this.get_name());

             & (rhs == null)) begin
             vm_status_container.scope.depth()) begin
             er.print_msg_object(this, rhs);

             in
             er.print_msg_object(this, rhs);
             port_info("MISCMP",
             sformatf("%0d Miscompare(s) for object %s@%0d vs. null",
             omparer.result,
             _m_uvm_status_container.scope.get(),
             this.get_inst_id()),
             __m_uvm_status_container.comparer.verbosity);
              1;



             & (comparer.compare_map.get(rhs) != null)) begin
             rer.compare_map.get(rhs) != this) begin
             er.print_msg_object(this, comparer.compare_map.get(rhs));

             ;  //don't do any more work after this case, but do cleanup


             & comparer.check_type && (rhs != null) &&
             e_name() != rhs.get_type_name())) begin
             status_container.stringv = { "lhs type = \"", get_type_name(),
                      "\" : rhs type = \"", rhs.get_type_name(), "\""};
             .print_msg(__m_uvm_status_container.stringv);



             begin
             .compare_map.set(rhs, this);
             field_automation(rhs, UVM_COMPARE, ""); // LINE 58-field macros
             compare(rhs, comparer);                 // LINE 59-do_compare()


             _status_container.scope.depth()==1)  begin
             status_container.scope.up();




                          (this, rhs);
                          == 0 && dc == 1);
```

69 Lines of code !!

It would be too complex to override the compare() base method !!

`uvm_object_utils(T)

macros/uvm_object_defines.svh



```
`define uvm_object_utils(T) \
  `uvm_object_utils_begin(T) \
  `uvm_object_utils_end
```

```
`define uvm_object_utils_begin(T)         \
 `m_uvm_object_registry_internal(T,T)\
 `m_uvm_object_create_func(T)             \
 `m_uvm_get_type_name_func(T)             \
 `uvm_field_utils_begin(T)
```

Register the transaction class with the factory

Define the `create()` method

Define the `get_type_name()` method

```
function void __m_uvm_field_automation (…)\
  begin \
    ... \
```

Defines first 20 lines of method: `__m_uvm_field_automation()`

Each field macro adds more code here

```
`define uvm_object_utils_end\
   end                      \
 endfunction                \
```

```
  end   \
endfunction \
```

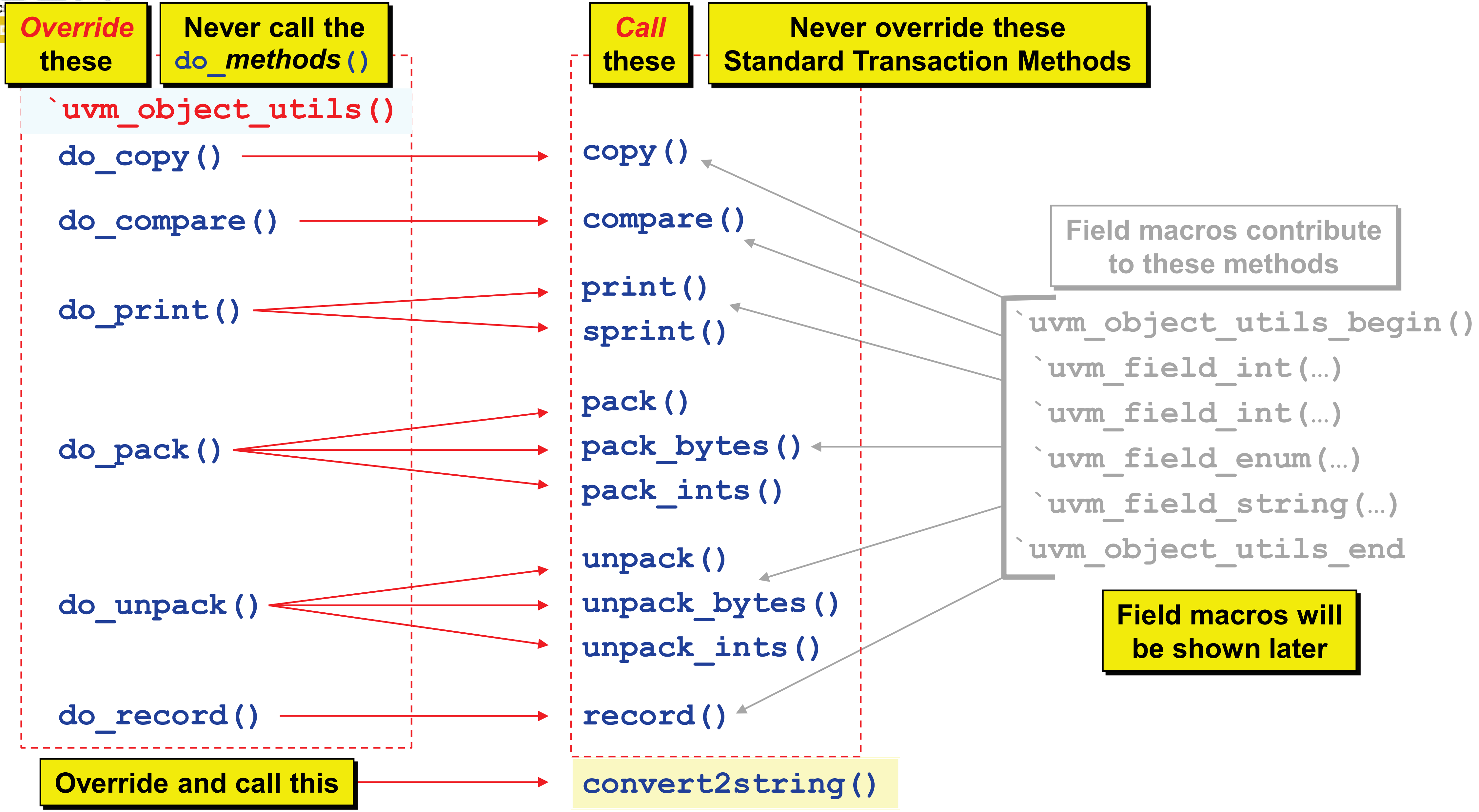
Defines last 2 lines of method: `__m_uvm_field_automation()`

Each `` `uvm_field_int `` adds 59 lines - big `case` statement

Overriding do_methods()

Defining Standard Transaction Method behavior using do_*methods*()

Standard Transaction Methods



Override these

Never call the do_*methods*()

`uvm_object_utils()

do_copy()

do_compare()

do_print()

do_pack()

do_unpack()

do_record()

Override and call this

Call these

Never override these Standard Transaction Methods

copy()

compare()

print()

sprint()

pack()

pack_bytes()

pack_ints()

unpack()

unpack_bytes()

unpack_ints()

record()

convert2string()

Field macros contribute to these methods

```
`uvm_object_utils_begin()
  `uvm_field_int(…)
  `uvm_field_int(…)
  `uvm_field_enum(…)
  `uvm_field_string(…)
`uvm_object_utils_end
```

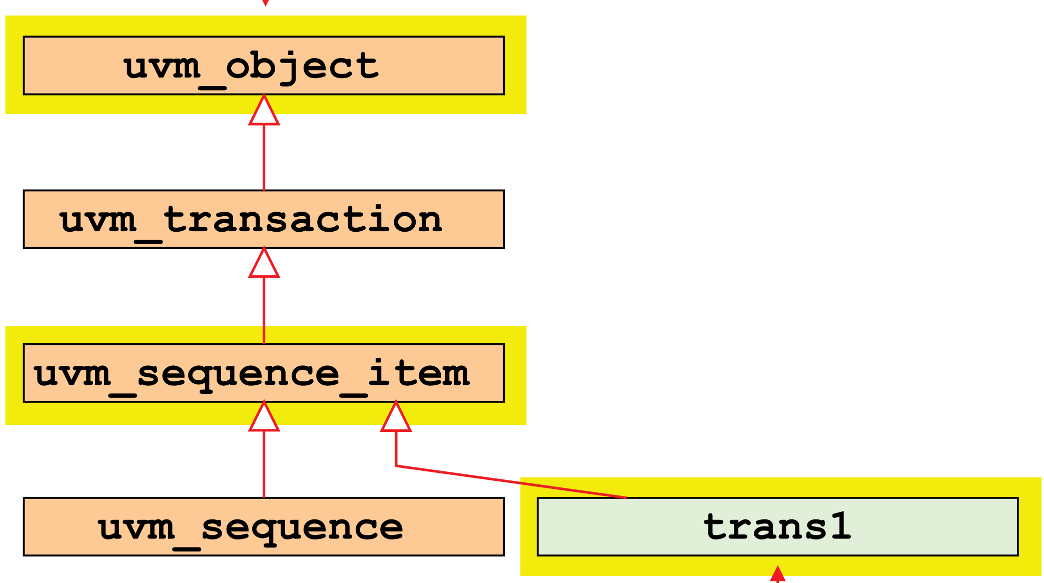
Field macros will be shown later

User-Defined Transaction Class

Derivative of uvm_object

```
class trans1 extends uvm_sequence_item;
  `uvm_object_utils(trans1)

        logic [15:0] dout;
  rand bit    [15:0] din;
  rand bit           ld, inc, rst_n;
  ...
  function void do_copy(uvm_object rhs);
    trans1 tr;
    if(!$cast(tr, rhs))
      `uvm_fatal("trans1", "FAIL: do_copy() cast");
    super.do_copy(rhs);
    dout  = tr.dout;
    din   = tr.din;
    ld    = tr.ld;
    inc   = tr.inc;
    rst_n = tr.rst_n;
  endfunction
  ...
endclass
```

uvm_object is the top-level base class in UVM



uvm_object

uvm_transaction

uvm_sequence_item

uvm_sequence

trans1

The user transaction type is a derivative of uvm_object

Transaction Class do_copy() Method

Upcasting & Downcasting

Assume trans1 tr1 object

```
class trans1 extends uvm_sequence_item;
  `uvm_object_utils(trans1)

      logic [15:0] dout;
  rand bit    [15:0] din;
  rand bit           ld, inc, rst_n;
  ...
  function void do_copy(uvm_object rhs);
    trans1 tr;
    if(!$cast(tr, rhs))
      `uvm_fatal("trans1", "...");
    super.do_copy(rhs);
    dout  = tr.dout;
    din   = tr.din;
    ld    = tr.ld;
    inc   = tr.inc;
    rst_n = tr.rst_n;
  endfunction
  ...
endclass
```

tr1.copy(t);

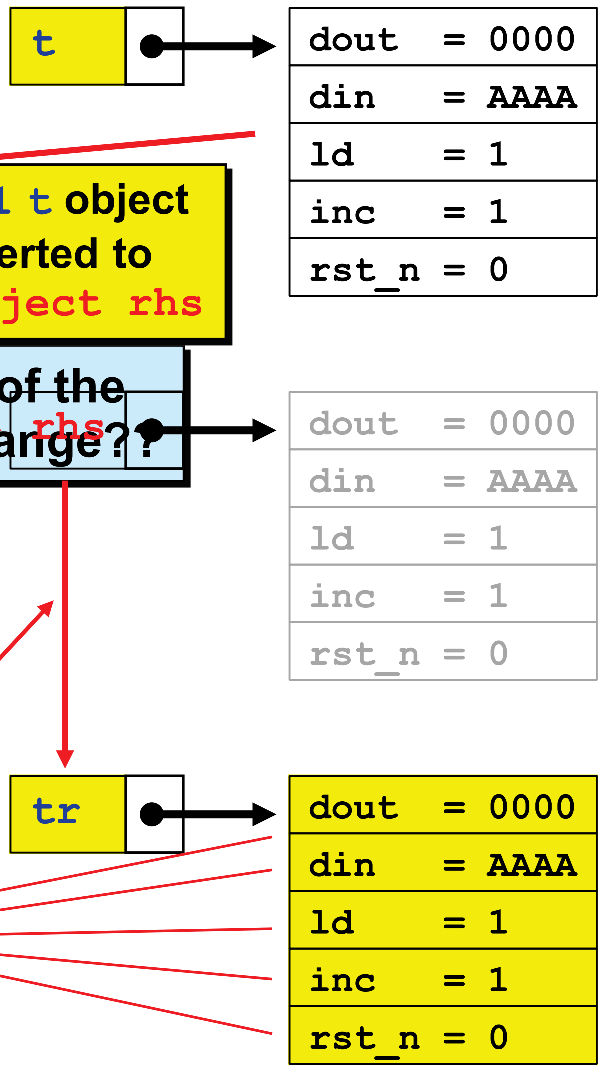
calls ...

do_copy(t);

Upcast

trans1 t object converted to uvm_object rhs

trans1 t object with five variables



Why do the first lines of the do_methods() look strange??

uvm_object rhs Cannot access variables

Downcast

Declare trans1 tr handle

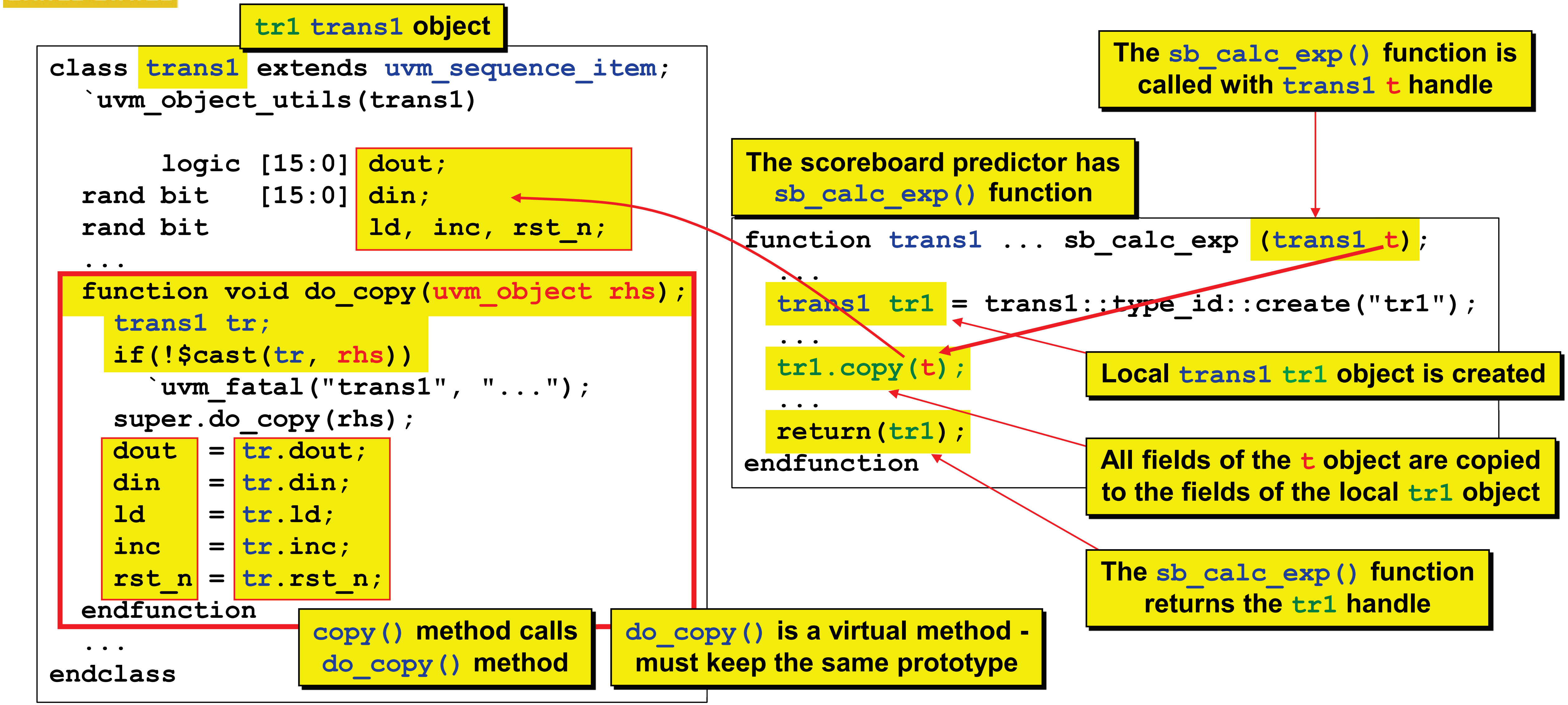
$cast uvm_object rhs handle to trans1 tr handle

do_copy() is a virtual meth must keep the same protot

Now copy tr signals to local tr1 trans1 signals

Transaction Class do_copy() Method

Example Usage from Scoreboard Predictor



tr1 trans1 object

```
class trans1 extends uvm_sequence_item;
  `uvm_object_utils(trans1)

       logic [15:0] dout;
  rand bit   [15:0] din;
  rand bit          ld, inc, rst_n;
  ...
  function void do_copy(uvm_object rhs);
    trans1 tr;
    if(!$cast(tr, rhs))
      `uvm_fatal("trans1", "...");
    super.do_copy(rhs);
    dout  = tr.dout;
    din   = tr.din;
    ld    = tr.ld;
    inc   = tr.inc;
    rst_n = tr.rst_n;
  endfunction
  ...
endclass
```

copy() method calls do_copy() method

do_copy() is a virtual method - must keep the same prototype

The sb_calc_exp() function is called with trans1 t handle

The scoreboard predictor has sb_calc_exp() function

```
function trans1 ... sb_calc_exp (trans1 t);
  ...
  trans1 tr1 = trans1::type_id::create("tr1");
  ...
  tr1.copy(t);
  ...
  return(tr1);
endfunction
```

Local trans1 tr1 object is created

All fields of the t object are copied to the fields of the local tr1 object

The sb_calc_exp() function returns the tr1 handle

Upcasting & Downcasting Variable Names

Avoid Confusing Names

Previous slide - We named the local `trans1` handle `tr`

- Many industry examples name the local transaction handle `rhs_`
- Using `rhs_` means that casting is done in the form `$cast(rhs_, rhs);`

This is confusing and therefore a poor practice

- Causes fields to be referenced as `rhs_.field_1` , …

Easy to confuse the `uvm_object` `rhs` handle with the transaction class `rhs_` handle

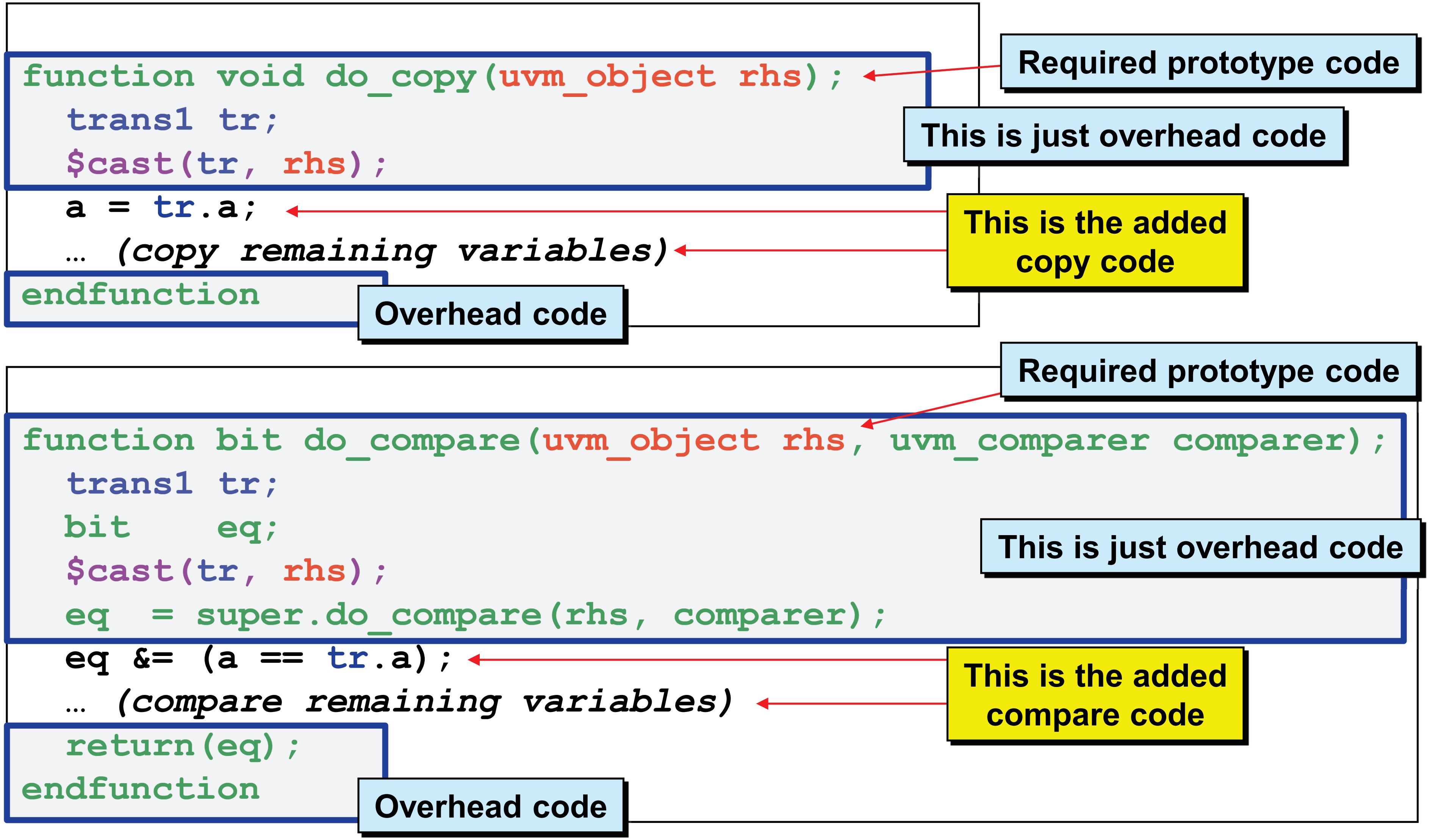
- Better practice: Use a transaction handle name like `tr`

Or another name that is visually distinct

Guideline: **Declare local transaction handles using distinct names such as `tr` and avoid local transaction handle names such as `rhs_`**

do_copy() & do_compare()

Template Methods



```
function void do_copy(uvm_object rhs);
  trans1 tr;
  $cast(tr, rhs);
  a = tr.a;
  … (copy remaining variables)
endfunction
```

Required prototype code

This is just overhead code

This is the added copy code

Overhead code

```
function bit do_compare(uvm_object rhs, uvm_comparer comparer);
  trans1 tr;
  bit    eq;
  $cast(tr, rhs);
  eq  = super.do_compare(rhs, comparer);
  eq &= (a == tr.a);
  … (compare remaining variables)
  return(eq);
endfunction
```

Required prototype code

This is just overhead code

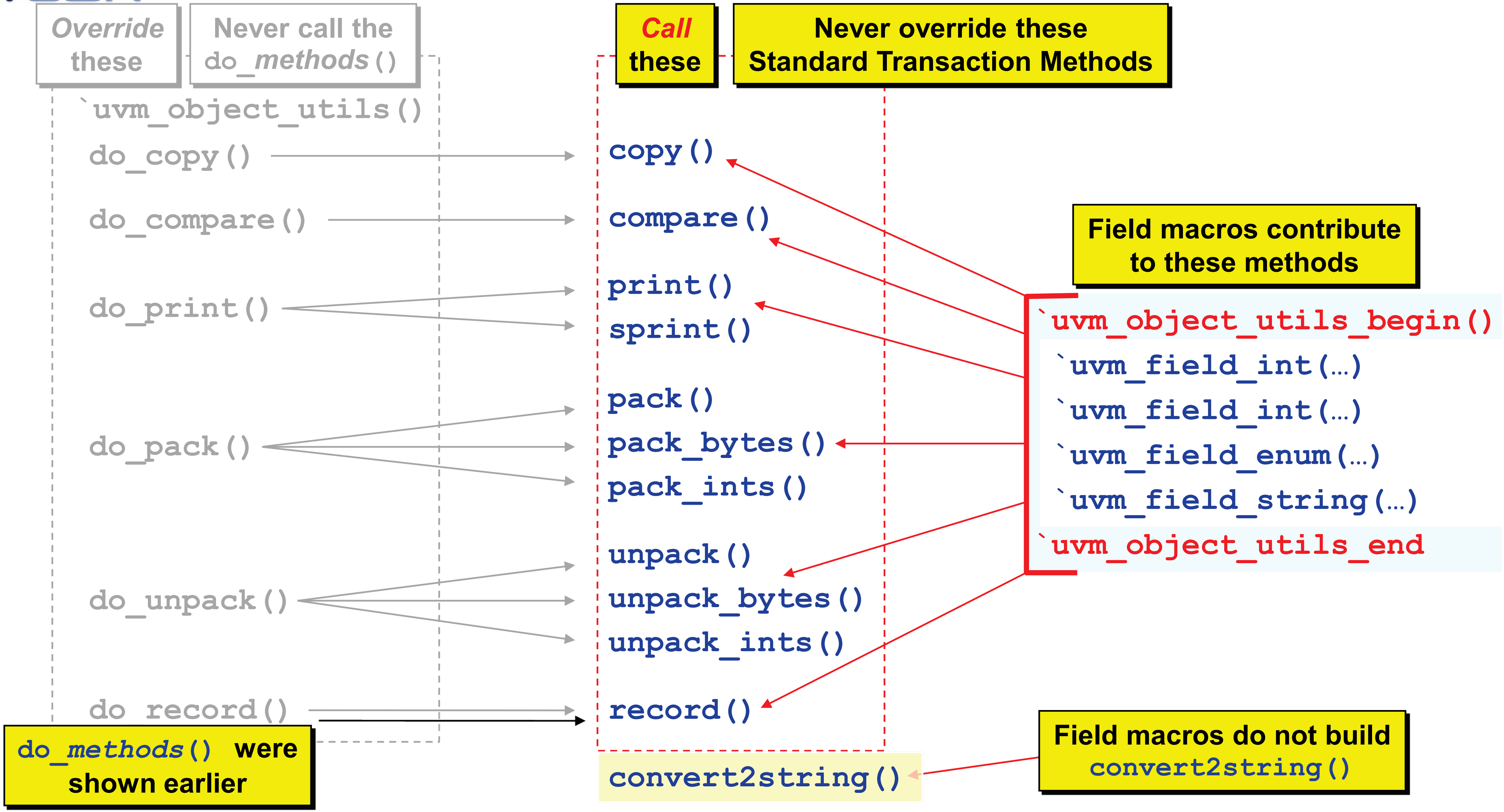
This is the added compare code

Overhead code

Using Field Macros

Standard Transaction Methods

Defining Standard Transaction Method behavior using field macros



Override these

Never call the do_*methods*()

```
`uvm_object_utils()
do_copy()
do_compare()
do_print()
do_pack()
do_unpack()
do_record()
```

***Call* these**

Never override these Standard Transaction Methods

```
copy()
compare()
print()
sprint()
pack()
pack_bytes()
pack_ints()
unpack()
unpack_bytes()
unpack_ints()
record()
convert2string()
```

Field macros contribute to these methods

```
`uvm_object_utils_begin()
  `uvm_field_int(...)
  `uvm_field_int(...)
  `uvm_field_enum(...)
  `uvm_field_string(...)
`uvm_object_utils_end
```

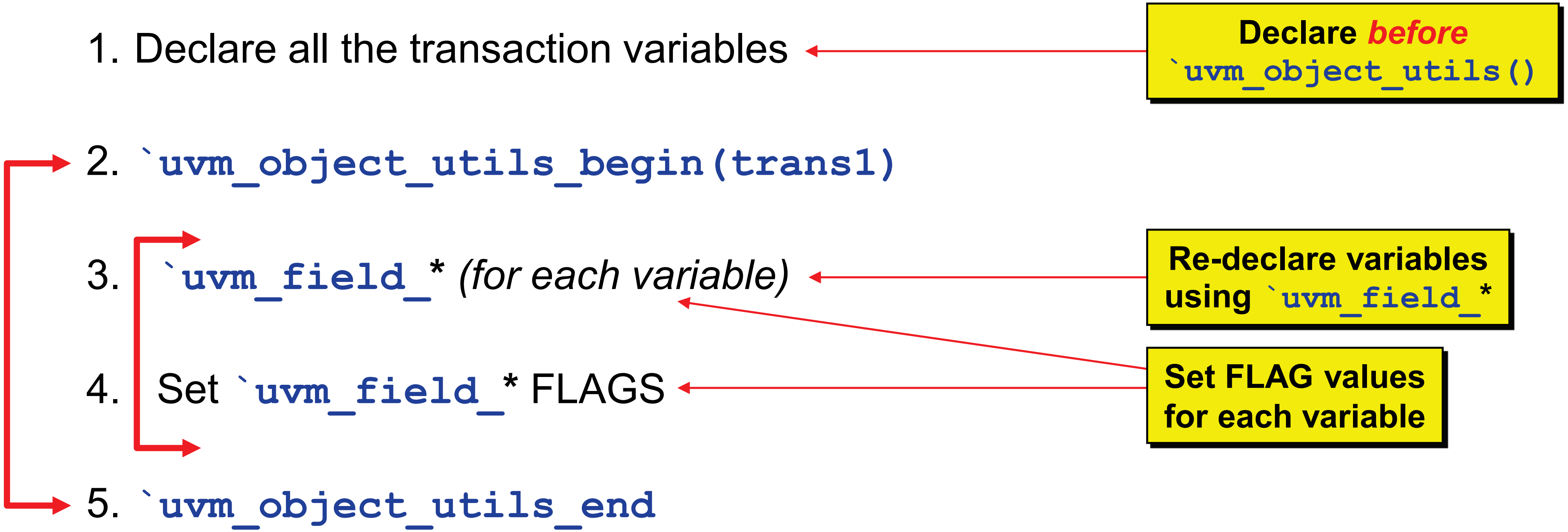
Field macros do not build `convert2string()`

`do_`*`methods`*`()` **were shown earlier**

Using Field Macros

Requirements

What is required to use field macros?



1. Declare all the transaction variables
2. `` `uvm_object_utils_begin(trans1) ``
3. `` `uvm_field_* `` *(for each variable)*
4. Set `` `uvm_field_* `` FLAGS
5. `` `uvm_object_utils_end ``

Declare *before* `` `uvm_object_utils() ``

Re-declare variables using `` `uvm_field_* ``

Set FLAG values for each variable

Transaction with Field Macros

Rules

```
class trans2 extends uvm_sequence_item;
  rand bit [7:0] a, b, c;

  `uvm_object_utils_begin(trans2)
    `uvm_field_int(a, UVM_ALL_ON)
    `uvm_field_int(b, UVM_ALL_ON)
    `uvm_field_int(c, UVM_ALL_ON)
  `uvm_object_utils_end
  ...
endclass
```

Same type and size: variables can be declared as a list

Same field macro flags: variables *MUST* be declared separately

```
    `uvm_field_int(  a,b,c,   UVM_ALL_ON)
```

ILLEGAL to group variables in the same field macro

```
    `uvm_field_int( {a,b,c}, UVM_ALL_ON)
```

STILL ILLEGAL
to concatenate variables in the same field macro

`uvm_field Macros

Data declaration field types

```
`uvm_field_int              (ARG,FLAG)
`uvm_field_enum           (T,ARG,FLAG)
`uvm_field_object           (ARG,FLAG)
`uvm_field_string           (ARG,FLAG)
`uvm_field_real             (ARG,FLAG)
`uvm_field_event            (ARG,FLAG)
```

int field macros are for any integral number-type

Includes most signals and buses (vectors)

Most commonly used field macros

```
`uvm_field_sarray_int       (ARG,FLAG)
`uvm_field_sarray_enum      (ARG,FLAG)
`uvm_field_sarray_object(ARG,FLAG)
`uvm_field_sarray_string(ARG,FLAG)
```

Static array field macros

```
`uvm_field_array_int        (ARG,FLAG)
`uvm_field_array_enum       (ARG,FLAG)
`uvm_field_array_object (ARG,FLAG)
`uvm_field_array_string (ARG,FLAG)
```

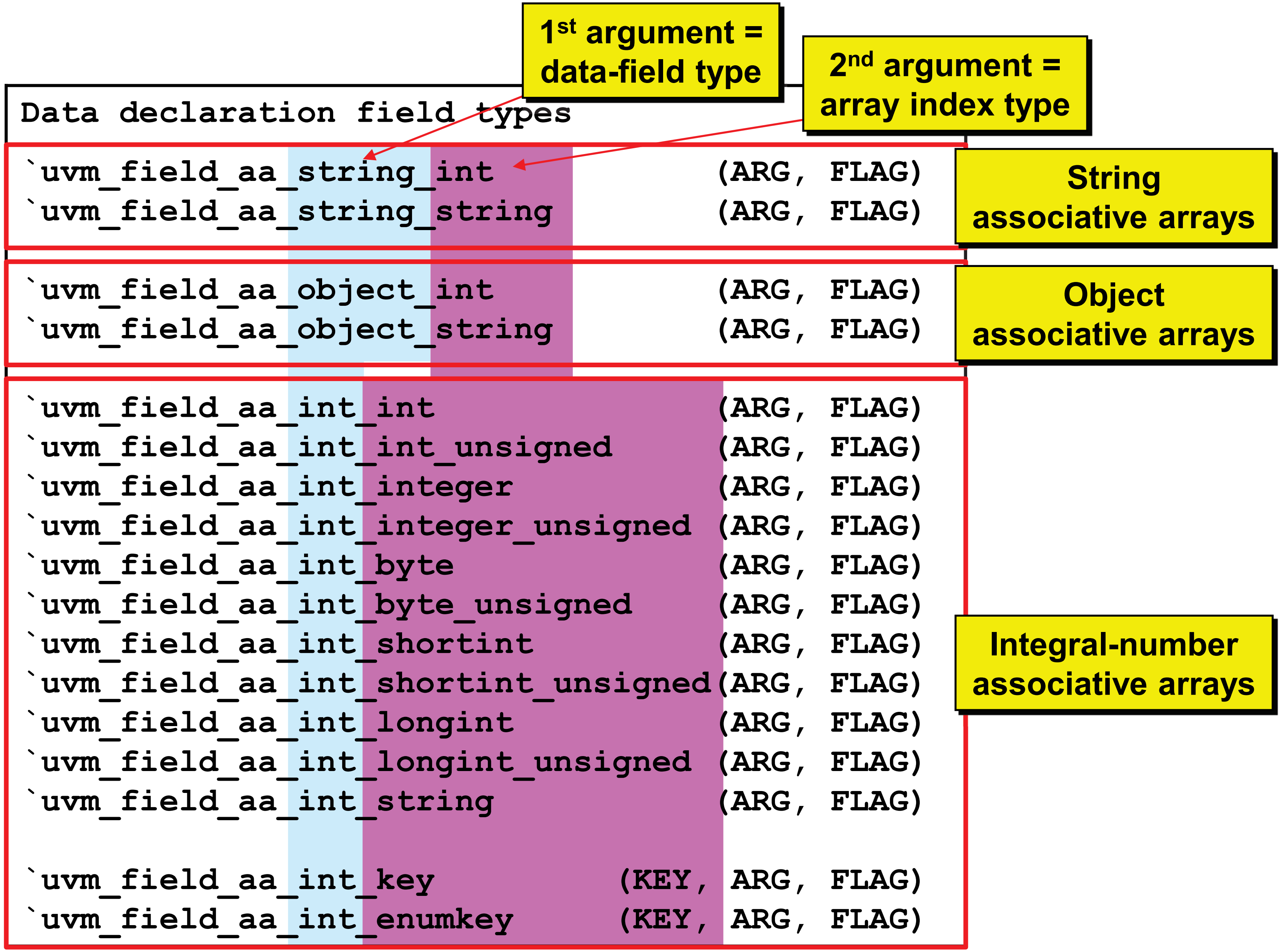
1-dimensional dynamic array field macros

```
`uvm_field_queue_int        (ARG,FLAG)
`uvm_field_queue_enum       (ARG,FLAG)
`uvm_field_queue_object (ARG,FLAG)
`uvm_field_queue_string (ARG,FLAG)
```

Queue field macros

`uvm_field Macros

Associative array field macros



1st argument = data-field type

2nd argument = array index type

```
Data declaration field types

`uvm_field_aa_string_int           (ARG, FLAG)
`uvm_field_aa_string_string        (ARG, FLAG)

`uvm_field_aa_object_int           (ARG, FLAG)
`uvm_field_aa_object_string        (ARG, FLAG)

`uvm_field_aa_int_int              (ARG, FLAG)
`uvm_field_aa_int_int_unsigned     (ARG, FLAG)
`uvm_field_aa_int_integer          (ARG, FLAG)
`uvm_field_aa_int_integer_unsigned (ARG, FLAG)
`uvm_field_aa_int_byte             (ARG, FLAG)
`uvm_field_aa_int_byte_unsigned    (ARG, FLAG)
`uvm_field_aa_int_shortint         (ARG, FLAG)
`uvm_field_aa_int_shortint_unsigned(ARG, FLAG)
`uvm_field_aa_int_longint          (ARG, FLAG)
`uvm_field_aa_int_longint_unsigned (ARG, FLAG)
`uvm_field_aa_int_string           (ARG, FLAG)

`uvm_field_aa_int_key         (KEY, ARG, FLAG)
`uvm_field_aa_int_enumkey     (KEY, ARG, FLAG)
```

String associative arrays

Object associative arrays

Integral-number associative arrays

UVM Field Macro Flags

Other macro flags on the next slide

- `UVM_ALL_ON` - Automatically creates the following important core data methods:

```
copy() & compare()
pack() & unpack()
record()
print() & sprint()
```

UVM Field Macro Flags

Multiple flags can be bitwise OR-ed together

- UVM Field Macro Flags

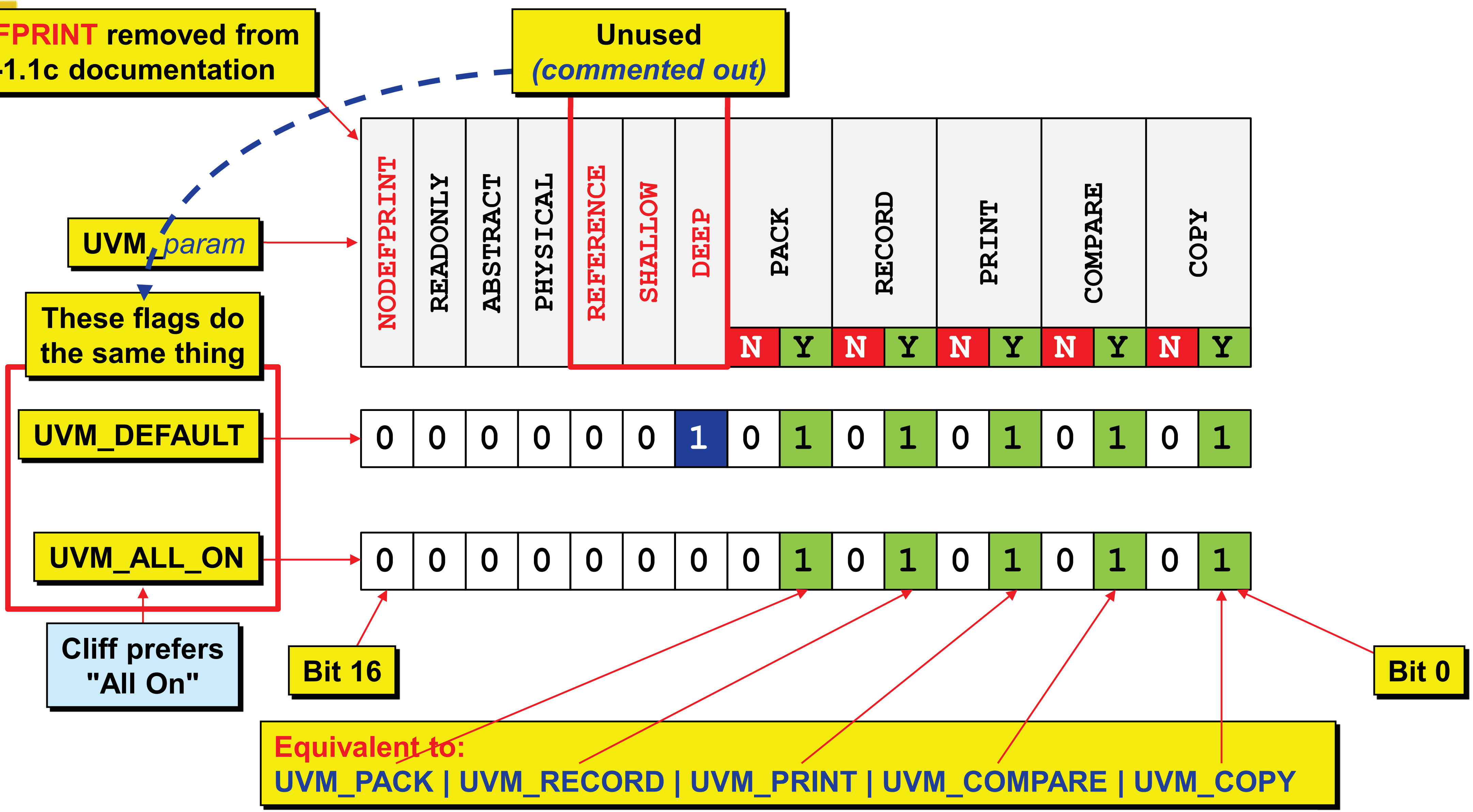
`UVM_ALL_ON`	Set all operations on (default)
`UVM_DEFAULT`	Use the default flag settings
`UVM_NOCOPY`	Do not copy this field
`UVM_NOCOMPARE`	Do not compare this field
`UVM_NOPRINT`	Do not print this field
`UVM_NODEFPRINT`	*(not documented in User Guide or Reference Manual)*
`UVM_NOPACK`	Do not pack or unpack this field
`UVM_PHYSICAL`	Treat as a physical field. Use physical setting in policy class for this field
`UVM_ABSTRACT`	Treat as an abstract field. Use the abstract setting in the policy class for this field
`UVM_READONLY`	Do not allow setting of this field from the set_*_local methods

Can also add the flags together but bitwise or'ed is safer ***(avoids double incrementing)***

Mentor warns about inefficiencies

Users like the ease of use

UVM Macro Flags



NODEFPRINT removed from uvm-1.1c documentation

Unused *(commented out)*

UVM_*param*

These flags do the same thing

	NODEFPRINT	READONLY	ABSTRACT	PHYSICAL	REFERENCE	SHALLOW	DEEP	PACK		RECORD		PRINT		COMPARE		COPY	
								N	Y	N	Y	N	Y	N	Y	N	Y
UVM_DEFAULT	0	0	0	0	0	0	1	0	1	0	1	0	1	0	1	0	1
UVM_ALL_ON	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	1

Cliff prefers "All On"

Bit 16

Bit 0

Equivalent to:
UVM_PACK | UVM_RECORD | UVM_PRINT | UVM_COMPARE | UVM_COPY

Field Macro Flags

Adding Multiple Flags

```
class trans3 extends uvm_sequence_item;
  rand bit [7:0] a, b, c;

  `uvm_object_utils_begin(trans3)
    `uvm_field_int(a, UVM_ALL_ON)
    `uvm_field_int(b, UVM_ALL_ON)
    `uvm_field_int(c, UVM_ALL_ON | UVM_NOCOPY)
  `uvm_object_utils_end

  function new (string name="trans3");
    super.new(name);
  endfunction

  `include "print_trans.sv"
endclass
```

Creates ALL standard transaction methods for these variables

Creates ALL standard transaction methods for this variable *EXCEPT* `copy()`

Legal Exception FLAGS:

`UVM_NOCOPY` `UVM_NOCOMPARE` `UVM_NOPRINT`
`UVM_NOPACK` `UVM_NORECORD`

OFF-FLAGS have precedence over ON-FLAGS

Adding Field Macro Flags

Multiple Flags Using | or +

Setting multiple flags with | separation is preferred

```
…
 `uvm_object_utils_begin(trans3)
   `uvm_field_int(a, UVM_ALL_ON)
   `uvm_field_int(b, UVM_ALL_ON)
   `uvm_field_int(c, UVM_NOCOPY | UVM_ALL_ON | UVM_NOCOPY)
 `uvm_object_utils_end
…
```

Mistakenly *OR-ing* UVM_NOCOPY twice still yields no-copy operation

Setting multiple flags with + separation is legal

```
…
 `uvm_object_utils_begin(trans3)
   `uvm_field_int(a, UVM_ALL_ON)
   `uvm_field_int(b, UVM_ALL_ON)
   `uvm_field_int(c, UVM_NOCOPY + UVM_ALL_ON + UVM_NOCOPY)
 `uvm_object_utils_end
…
```

Mistakenly *adding* UVM_NOCOPY twice clears the no-copy bit

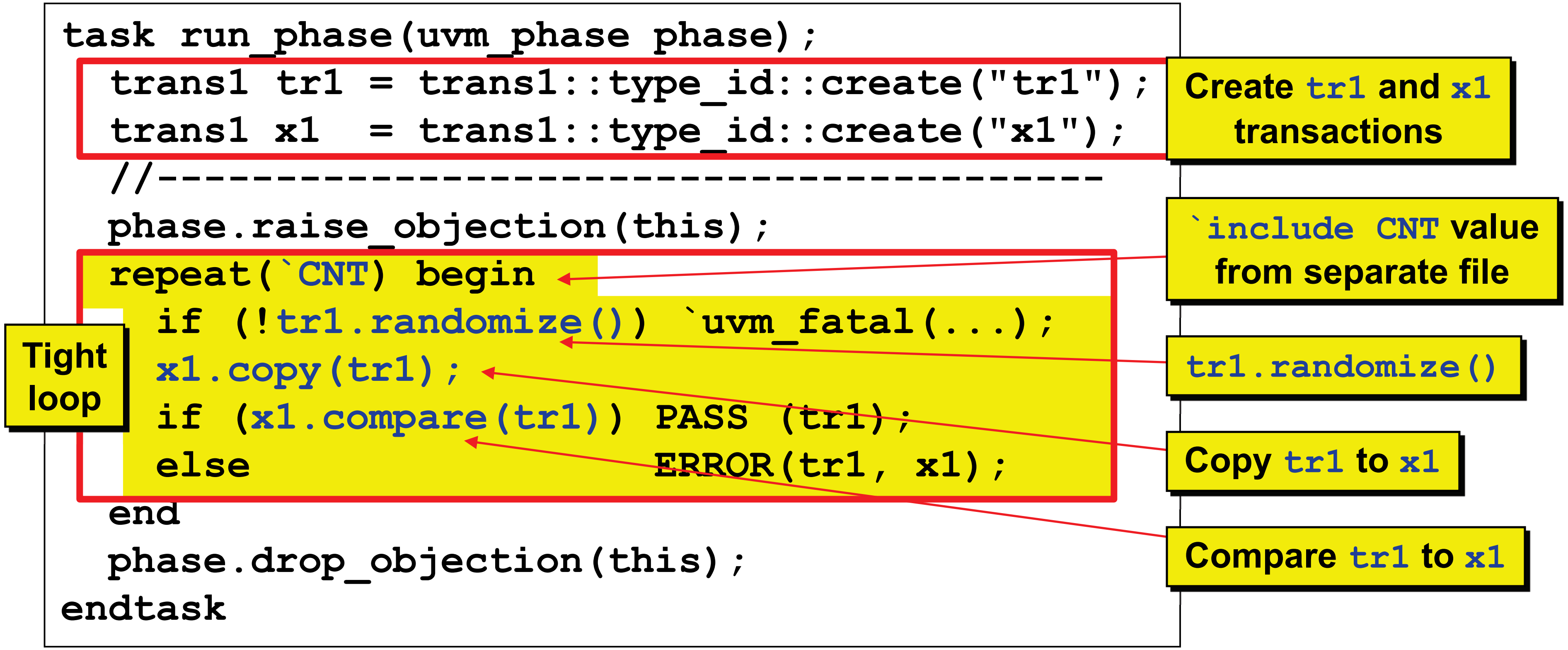
c variable will be copied

Efficiency Benchmarks

Benchmarking Methodology

From test1.sv File

- `test1` component with a tight loop:
  - Transactions repeatedly: (1) `randomize()` (2) `copy()` (3) `compare()`



```
task run_phase(uvm_phase phase);
  trans1 tr1 = trans1::type_id::create("tr1");
  trans1 x1  = trans1::type_id::create("x1");
  //-------------------------------------------
  phase.raise_objection(this);
  repeat(`CNT) begin
    if (!tr1.randomize()) `uvm_fatal(...);
    x1.copy(tr1);
    if (x1.compare(tr1)) PASS (tr1);
    else                 ERROR(tr1, x1);
  end
  phase.drop_objection(this);
endtask
```

Tight loop

Create `tr1` and `x1` transactions

`` `include CNT `` value from separate file

`tr1.randomize()`

Copy `tr1` to `x1`

Compare `tr1` to `x1`

Benchmarking Methodology

From trans1.sv Files

- How to setup transactions is always tricky
- `trans1` transactions benchmarks:
  - 5 `rand` inputs
  - 5 `rand` outputs
  - 5 non-`rand` outputs
  - `do_copy()` & `do_compare()`
  - Field macros
  - `do_copy()` with & without `super.do_copy()`
  - `do_compare()` with & without `super.do_compare()`

All benchmark code is in Annex B of the paper

You can try it!

All inputs randomized

Penalty for unnecessary randomization of outputs??

Penalty for using field macros??

Penalty for unnecessary calls to `super`-base methods??

Benchmark Results

2018 Benchmarks

Penalty Benchmark	Simulator A	Simulator B	Simulator C
	CNT=100M	CNT=100M	CNT=100M
Unnecessary rand-outputs -vs- non-randomized outputs *(Using do_methods())*	16.5% slower	11.3% slower	13.8% slower
Unnecessary rand-outputs -vs- non-randomized outputs *(Using Field Macros)*	12.1% slower	5.3% slower	11.8% slower
Penalty for using Field Macros -vs- using do_*methods*()	6.0% slower	13.8% slower	3.9% slower
Penalty for calling unnecessary super.do_*methods*()	2.4% slower	3.3% slower	2.2% slower

Do NOT randomize transaction output fields

Using Field Macros has a penalty

Calling `super.do_methods()` has a small-ish penalty

UVM Basic Transaction Objects

- On the next slides, we will build:
  - `trans1` ← Basic transaction type built from `uvm_sequence_item`
  - `read_sequence`, `write_sequence` ← Example `uvm_sequence` code
  - `write_read` ← Example sequence of sequences

UVM Transaction

(Built from uvm_sequence_item)

Extend uvm_sequence_item to build the base transaction

```
class trans1 extends uvm_sequence_item;
  `uvm_object_utils(trans1)

  rand bit     rw_n, cs_n;
  rand data_t data;
  rand addr_t addr;

  typedef enum {READ, WRITE} rw_e;
  rand rw_e rw_type;

  constraint c1 {(rw_type == READ ) -> rw_n == '1;
                 (rw_type == WRITE) -> rw_n == '0;}

  function new(string name="trans1");
    super.new(name);
  endfunction

  ...
```

Register the trans1 object in the UVM factory

NOTE: `uvm_object_utils *NOT* `uvm_sequence_utils

Randomizable data members

Randomization "knob"

Randomization constraints

Common transaction constructor *(no parent)*

Optional: add convert2string() and post_randomize() methods *(next slide)*

Guideline: create transactions by extending uvm_sequence_item *(it is common to create sequences of transactions)*

UVM Transaction

Add convert2string() & post_randomize()

`class trans1` **(cont.)**

```
  ...


  function string convert2string();
    return($sformatf(" addr=%3h  data=%2h  rw_n=%b  cs_n=%b",
                     addr, data, rw_n, cs_n)});
  endfunction

  function void post_randomize();
    `uvm_info("trans1", this.convert2string(), UVM_HIGH);
  endfunction
endclass
```

Returns a formatted string for this object

Prints the formatted string after `randomize()`

Sequence: read_sequence

(Read sequence definition)

Extend uvm_sequence to build a sequence of transactions

uvm_sequence is a parameterized class *(passes trans1 transactions)*

```
class read_sequence extends uvm_sequence #(trans1);
  `uvm_object_utils(read_sequence)

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

  task body;
    trans1 tr;
    tr = trans1::type_id::create("tr");
    //-----------------------------------------------
    start_item (tr);
    if (!(tr.randomize() with {rw_type==READ;}))
        `uvm_error("RAND", "Failed randomization")
    finish_item (tr);
  endtask
endclass
```

Register the read_sequence in the UVM factory *(object type)*

Common constructor

body method

Standard steps:

(1) declare a transaction (tr)

(2) create *(register)* the tr in the factory

(3) start communication with the sequencer

(4) randomizes the tr data with added constraint (READ sequence)

(5) finish communication with the sequencer

Sequence: write_sequence

(Write sequence definition)

Extend uvm_sequence to build a sequence of transactions

uvm_sequence is a parameterized class *(passes trans1 transactions)*

```
class write_sequence extends uvm_sequence #(trans1);
  `uvm_object_utils(write_sequence)

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

  task body;
    trans1 tr;
    tr = trans1::type_id::create("tr");
    //----------------------------------------
    start_item (tr);
    if (!(tr.randomize() with {rw_type==WRITE;}))
        `uvm_error("RAND", "Failed randomization")
    finish_item (tr);
  endtask
endclass
```

Register the write_sequence in the UVM factory *(object type)*

Common constructor

body method

Standard steps:

(1) declare a transaction (tr)

(2) create *(register)* the tr in the factory

(3) start communication with the sequencer

(4) randomizes the tr data with added constraint (WRITE sequence)

(5) finish communication with the sequencer

Sequence: write_read

(sequence defined using other sequences)

Extend uvm_sequence to build a *sequence of sequences*

```
class write_read extends uvm_sequence #(trans1);
  `uvm_object_utils(write_read)

  rand int cnt;
  constraint loop_cnt {cnt inside {[3:5]};}

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

  task body;
    write_sequence wseq;
    read_sequence  rseq;
    wseq = write_sequence::type_id::create("wseq");
    rseq =  read_sequence::type_id::create("rseq");
    //----------------------------------------------
    repeat (cnt) begin
      wseq.start(m_sequencer);
      rseq.start(m_sequencer);
    end
  endtask
endclass
```

Common constructor

uvm_sequence is a parameterized class *(passes trans1 transactions)*

Register the write_read in the UVM factory

Setup and constrain randomizable cnt

Standard steps:
Declare and create write_sequence(wseq) and read_sequence(rseq)

Randomized repeat(cnt)

Start write_sequence(wseq) on m_sequencer

Start read_sequence(rseq) on m_sequencer

`uvm_do Macros

`uvm_do sequence or sequence item

`uvm_do actions

		Macro Inputs				UVM actions			
		SEQ_OR_ITEM	SEQUENCER	PRIORITY	{CONSTRAINTS}	create()	start_item()	randomize()	finish_item()
Common	`uvm_do(I)	X				X	X	X	X
	`uvm_do_with(I,{C})	X			X	X	X	X	X
Common vsequencer	`uvm_do_on(I,S)	X	X			X	X	X	X
	`uvm_do_on_with(I,S,{C})	X	X		X	X	X	X	X
Less Common	`uvm_do_pri(I,P)	X		X		X	X	X	X
	`uvm_do_pri_with(I,P,{C})	X		X	X	X	X	X	X
	`uvm_do_on_pri(I,S,P)	X	X	X		X	X	X	X
	`uvm_do_on_pri_with(I,S,P,{C})	X	X	X	X	X	X	X	X

Summary of Rules

- `do_`*methods*`()` rule: you must use `` `uvm_object_utils() ``
- Field macros rule: declare the transaction variables before calling field macros
- Field macros rule: declare variables before registering the transaction with the factory
- Field macros rule: you must use:

  `` `uvm_object_utils_begin() / `uvm_object_utils_end ``
- Field macros rule: each variable in a separate field macro

Variables cannot be grouped into a common field macro definition

Summary of Important Guidelines

- Guideline: do not directly override standard trans methods

  `copy()`, `compare()`, etc.

  Get a life !!

- Guideline: never manually implement the `create()` method

  Call `` `uvm_object_utils() `` to automatically implement `create()`

- Guideline: Transactions should include a `convert2string()` method

  Always !!

- Guideline: Avoid using the `print()` and `sprint()` methods

  The outputs are verbose

- Guideline: *If you must,* use `sprint()` over `print()`

`sprint()` can be called from messaging macros

Better yet … use `convert2string()`

`convert2string()` is more simulation and more print-space efficient

Thank you!

Please continue with Part 2

IEEE 1800.2 UVM - Changes Useful UVM Tricks & Techniques

Part 2

Clifford E. Cummings

World Class Verilog, SystemVerilog & UVM Training

1639 E 1320 S, Provo, UT 84606
Voice: 801-960-1996
Email: cliffc@sunburst-design.com
Web: www.sunburst-design.com

Life is too short for bad or boring training!

Connect with Cliff on Linked in

UVM Basic Message Commands

Same techniques apply to OVM

Good reference paper:
UVM Message Display Commands - Capabilities, Proper Usage and Guidelines
www.sunburst-design.com/papers/CummingsSNUG2014AUS_UVM_Messages.pdf

Introduction

Why the UVM messages paper ??

- UVM verbosity settings are ***NOT*** message priority settings!

UVM Verbosity ≠ Message Priority !!

UVM Verbosity = *!(Message Priority)*

- **UVM_LOW** is not a low priority message
- **UVM_LOW** is one of the highest priority messages !!

- Reference sources and public examples … *get it wrong !!*

UVM User Guide
UVM Class Reference
+2 recent UVM books

- The paper offers guidelines on proper usage
- The paper shows useful messaging tricks

UVM Basic Printing Guidelines

- Printing command types
  - Verilog `$display` commands
  - Messages & messaging macros
  - `UVM_LOW`
  - `convert2string`

Guideline: quit using `$display`
(quit using `$display` / `$write` / `$strobe`)

Guideline: replace `$display` commands with:
`` `uvm_info("``*id*`", "`*msg*`", UVM_MEDIUM)`

`UVM_LOW` should almost *NEVER* be used

Widely misused in books and examples

User-defined formatting
(like `$display`)

Guideline: override `convert2string` method in all data/transaction classes

`convert2string` becomes a built-in *"show_my_contents"* method

UVM Message Facilities

Good messaging reference:
A Practical Guide to Adopting the Universal Verification Methodology (UVM)
Rosenberg & Meade

- `$display` - does not allow easy message filtering

- `uvm_report_info/fatal`* methods allow message filtering
  - by id -or-
  - by verbosity settings

`uvm_report_info/fatal`* methods include:

```
uvm_report_info   (...)
uvm_report_warning(...)
uvm_report_error  (...)
uvm_report_fatal  (...)
```

- `` `uvm_info/fatal``* macros:
  - Further simplify usage of `uvm_report_info/fatal`*
  - Include automatic file and line number reporting
  - Are more simulation efficient than `uvm_report_info/fatal` * methods

`` `uvm_info/fatal``* macros include:

```
`uvm_info   (...)
`uvm_warning(...)
`uvm_error  (...)
`uvm_fatal  (...)
```

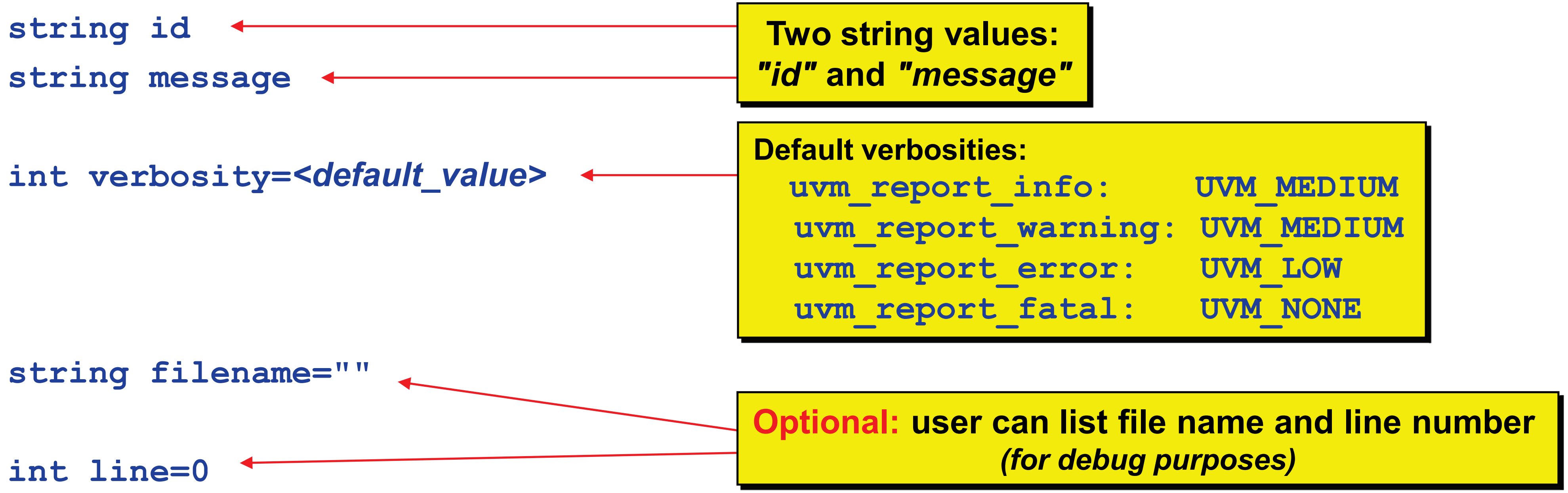
These macros recommended by all vendors

Macros avoid `$sformat` processing

uvm_report_info/fatal* Messages

- UVM has reporting services built into all **`uvm_component`**(s)

- UVM messages take up to 5 arguments (last 3 have defaults)



```
string id
string message

int verbosity=<default_value>

string filename=""

int line=0
```

Two string values:
"id"* and *"message"

Default verbosities:

uvm_report_info:	UVM_MEDIUM
uvm_report_warning:	UVM_MEDIUM
uvm_report_error:	UVM_LOW
uvm_report_fatal:	UVM_NONE

Optional: user can list file name and line number
(for debug purposes)

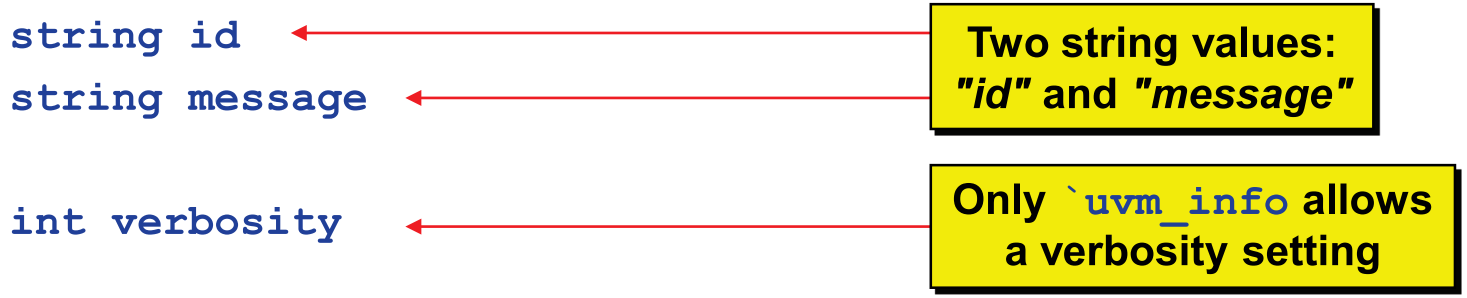
```
task run;
  uvm_report_info("run", "env still running", UVM_HIGH);
endtask
```

`uvm_info/fatal* Macros

- UVM macros are more simulation efficient than messages

Explanation on the next slide

- UVM macros take 2-3 arguments, depending on macro type



```
string id
string message

int verbosity
```

Two string values: *"id"* and *"message"*

Only `` `uvm_info `` allows a verbosity setting

Default macro verbosities that cannot be changed:

```
`uvm_warning: UVM_NONE
`uvm_error:   UVM_NONE
`uvm_fatal:   UVM_NONE
```

Macros automatically include *file name* and *line number* *(good for debugging)*

```
task run;
  `uvm_info("run", " env still running", UVM_HIGH)
endtask
```

UVM Messaging Macro Advantages

- UVM message macros:
  - Are more simulation efficient
  - Include `` `__FILE__ `` and `` `__LINE__ `` arguments
  - `` `uvm_warning `` / `error` / `fatal` include pre-defined default `UVM_VERBOSITY` settings

Wraps `uvm_report_*` calls in an `if`-statement

Removes expensive string processing if the verbosity setting would exclude the `uvm_report_*` calls

More efficient than `uvm_report` methods

SystemVerilog-2009

Automatically reports file and line numbers - good for debugging

To turn off `FILE` and `LINE` info

During Compilation: **use command line switch `+define+UVM_REPORT_DISABLE_FILE_LINE`**

Avoids new-user mistakes
(like setting `uvm_report_error` verbosity to `UVM_HIGH`)

convert2string()

Default returns ""

- **convert2string()** is a virtual function defined in **uvm_object**

- **convert2string()** is user-defined in the data/transaction class
  - This virtual function is a user-definable hook
  - Called directly by the user
  - *Users* provide object info in the form of a string
  - No **uvm_printer** policy object required
  - Format is fully user-customizable

From uvm_object base class

Simple & simulation efficient

Fields declared in `uvm_field_* macros *will not automatically appear* in calls to convert2string()

Unlike sprint

Good for applications that do not require consistent formatting offered by: print / sprint / do_print

Guideline: add convert2string() to all data/transaction classes

$sformat, $sformatf & $psprintf Commands

What Are The Differences?

- `$sformat` is used to generate a formatted string

Stand-alone command

String to be written

Formatted string or variables that represent strings

Arguments that satisfy format specifiers

```
$sformat (string_var, "formatted_string" [, list_of_arguments]);
```

(Can be integral or unpacked array of byte)

- `$sformatf` behaves like `$sformat` except:
  - Function that returns a string
  - Therefore - no first ***string_var*** argument

Function that returns string

Formatted string or variables that represent strings

Arguments that satisfy format specifiers

```
string_var = $sformatf ("formatted_string" [, list_of_arguments]) ;
```

NON-standard

- `$psprintf` - same as `$sformatf`

Appears to be implemented by all vendors

UVM Message Verbosity

- What is verbosity?
  - *Highly verbose* simulations would show lots of messages
  - *Minimally verbose* simulations would only show important messages

```
<sim_cmd>  +UVM_VERBOSITY=UVM_HIGH
```

500 = UVM_DEBUG	Print if selected verbosity is UVM_DEBUG
400 = UVM_FULL	Print if selected verbosity is UVM_FULL or lower
300 = UVM_HIGH	Print if selected verbosity is UVM_HIGH or lower
200 = UVM_MEDIUM	Print if selected verbosity is UVM_MEDIUM or lower
100 = UVM_LOW	Print if selected verbosity is UVM_LOW or lower
0 = UVM_NONE	Print always

Cannot be disabled by verbosity level setting

```
<sim_cmd>  +UVM_VERBOSITY=UVM_DEBUG
```

gher ve

Run-time command - run with a different verbosity *without recompiling!*

UVM Message Verbosity

Equivalent Verbosity Values

- UVM built-in **`uvm_verbosity`** enumerated values:

```
UVM_DEBUG  = 500
UVM_FULL   = 400
UVM_HIGH   = 300
UVM_MEDIUM = 200
UVM_LOW    = 100
UVM_NONE   = 0
```

Prints level 500 and lower

Prints level 200 and lower

- Two ways to change the verbosity for debugging:

```
<sv_sim_cmd> +UVM_VERBOSITY=UVM_LOW
```

Does not require re-compilation

```
set_report_verbosity_level_hier(UVM_LOW);
```

Can be put in a test

Useful Debugging Trick

Testbench & Factory Debugging

Unconditional Printing

- Good technique to view testbench and factory setup

```
class test_base extends uvm_test;
  ...
  uvm_factory factory=uvm_factory::get();

  function void start_of_simulation_phase(uvm_phase phase);
    super.start_of_simulation_phase(phase);
    this.print();
    factory.print();
  endfunction

  ...
endclass
```

`start_of_simulation` **phase**
(after the testbench is built and connected)

Add this code to print out the testbench structure

Add this code to print out the factory entries and overrides

PROBLEM: these printouts are unconditional
(not controlled by verbosity)

Could use `` `uvm_info(… this.sprint() …) ``

There is no `factory.sprint()`

Testbench & Factory Debugging

Verbosity-Controlled Printing

Cool Trick

- ***Better*** technique to view testbench and factory setup

```
class test_base extends uvm_test;
  ...
  uvm_factory factory=uvm_factory::get();

  function void start_of_simulation_phase(uvm_phase phase);
    super.start_of_simulation_phase(phase);

    if (uvm_report_enabled(UVM_HIGH)) begin
      this.print();
      factory.print();
    end
  endfunction
  ...
endclass
```

`start_of_simulation` phase
(after the testbench is built and connected)

Conditionally **execute `*.print()` commands when verbosity= `UVM_HIGH` or higher**

Print testbench structure and factory entries

Allows conditional printing based on verbosity

UVM Documentation Errors

Existing Documentation Problems

- UVM_LOW is pervasive in References, Books & Examples
  - **UVM User Guide**
    - Uses `$display` once
    - Uses 3 `` `uvm_info `` macros with bugs in the examples
    - Uses 5 `` `uvm_info `` macro examples with `UVM_LOW` - wrong verbosity
    - Uses 2 `` `uvm_info `` macro examples without `UVM_LOW` - correct!
  - **UVM Class Reference**
    - Uses 1 `` `uvm_info `` macro with bugs in the example
    - Uses 3 `` `uvm_info `` macro examples with `UVM_LOW` - wrong verbosity
    - Uses 2 `` `uvm_info `` macro examples without `UVM_LOW` - correct!
  - **Popular UVM Book published in 2013**
    - More than 20 examples improperly use `UVM_LOW`
  - **Popular UVM Beginner's Guide published in 2013**
    - More than 30 examples improperly use `UVM_LOW`

No wonder the UVM books get it wrong!

For low-priority messages

Summary of Important Guidelines

Sunburst Design Usage Guidelines

Think of `` `uvm_info `` as your new `$display` command

Macro Type/Verbosity	Usage Guideline
`` `uvm_fatal (...) ``	***fatal*** **- test-aborting errors**
`` `uvm_error (...) ``	**non-aborting simulation *errors***
`` `uvm_warning (...) ``	***error-inject warnings***
`` `uvm_info (... UVM_NONE) ``	**for final reports**
`` `uvm_info (... UVM_LOW) ``	**high priority messages**
`` `uvm_info (... UVM_MEDIUM) ``	**normal messages - replaces `$display`**

Non-maskable*

Use sparingly!

Almost **always prints**

---------------------------- *Above messages print by default* ----------------------------

`` `uvm_info (... UVM_HIGH) ``	**(1) passing transactions** **(2) conditionally print testbench & factory info**
`` `uvm_info (... UVM_FULL) ``	**print UVM status messages**
`` `uvm_info (... UVM_DEBUG) ``	**add debug messages**

Almost **always *OFF***

Section Agenda

Using UVM Analysis Ports & Paths

- Basic queues, mailboxes and TLM FIFOs ← **1st pass**
- Subscriber satellite TV analogy
- Analysis paths & analysis ports, exports, and imps
- TLM FIFOs ← **More detail**
- Importance of the `copy()` method
- How analysis port connections work - `write()` method
- Summary & Conclusions

The paper has more details and more examples

UVM Analysis Port Functionality and Using Transaction Copy Commands
`www.sunburst-design.com/papers/CummingsSNUG2018AUS_UVMAnalysisCopy.pdf`

Important SystemVerilog Features

PAY ATTENTION !!

Queues will be used to store *component* handles

- Queues
  - `push_back()` method to put a handle into the queue
  - `foreach()` method to walk through all stored handles
  - Does not have blocking `get()` method

Can store class handles - *great for storing connected components*

Not too useful for scoreboards

Mailboxes will be used to store *transaction* handles

- Mailboxes
  - Has nonblocking `try_put()` method
  - Has blocking `get()` method

Can store class handles - *great for storing transactions*

Important for scoreboards

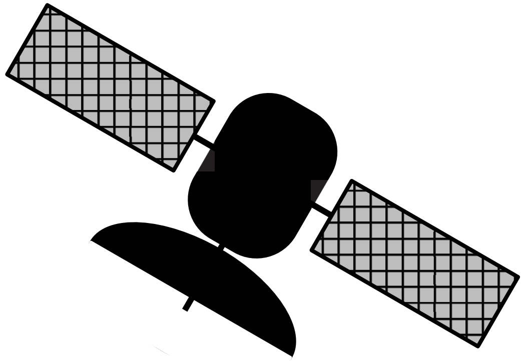
- Analysis path considerations:
  - Must start with `uvm_analysis_port` and end with `uvm_analysis_imp`
  - `uvm_tlm_fifo` *cannot* terminate an analysis path
  - `uvm_tlm_analysis_fifo` *CAN* terminate an analysis path

Must include `write()` method

Built using mailboxes

Very useful for scoreboards !!

Subscriber Satellite TV Analogy



- Two ways to watch a broadcast satellite TV program
  - Watch the program live
  - Record the program to a DVR to view later
- Satellite programs are broadcast as scheduled
- No way to restart a broadcast program
- Subscribers not allowed to change the live program

There might be 1,000's of viewers

There might be *NO* viewers

No way to communicate back to the satellite

Other viewers would object to restarting the program

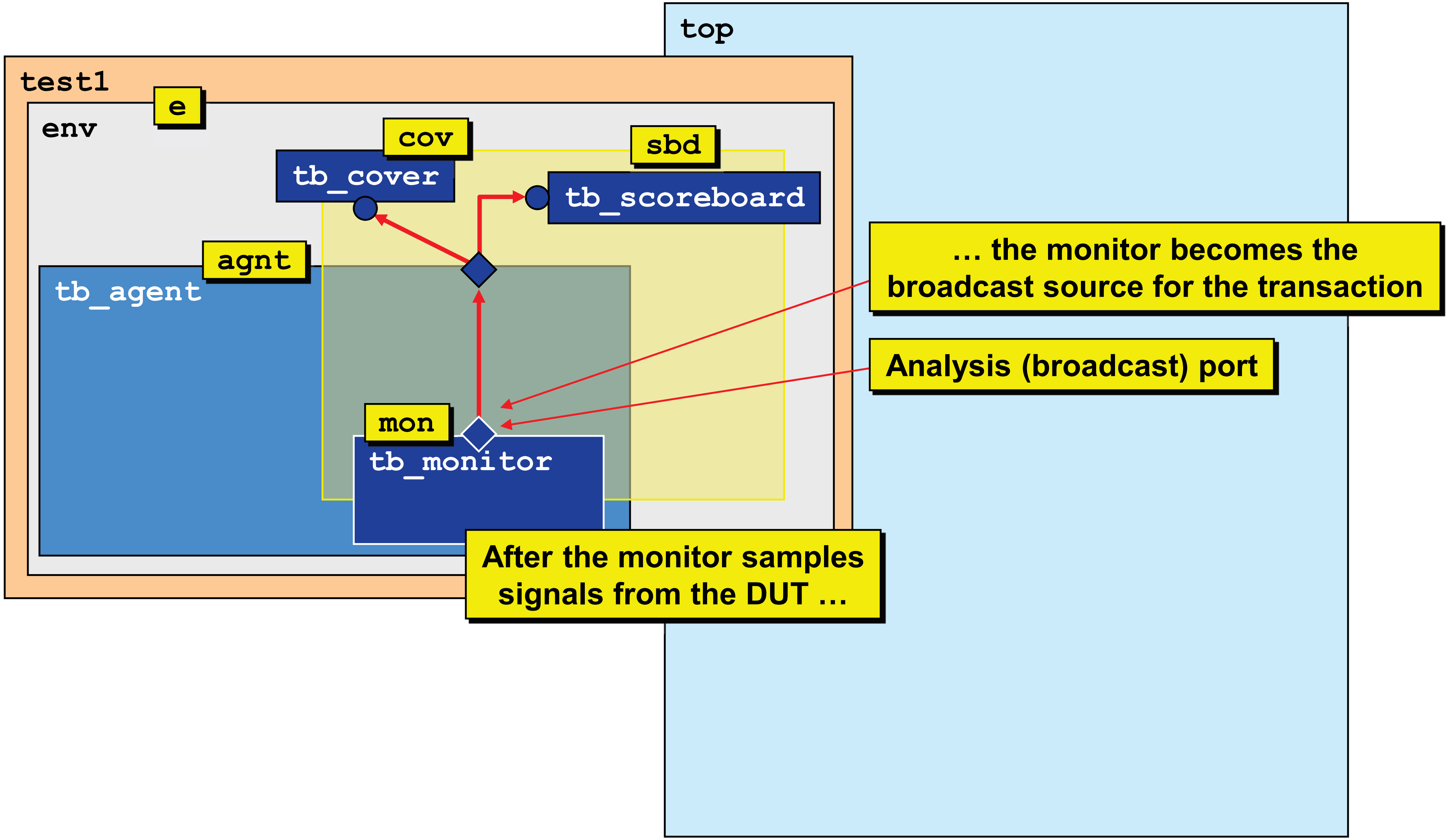
With the right equipment, you can modify your copy

Analysis Port Connections

and TLM FIFOs

Common UVM Components

Overview Block Diagram



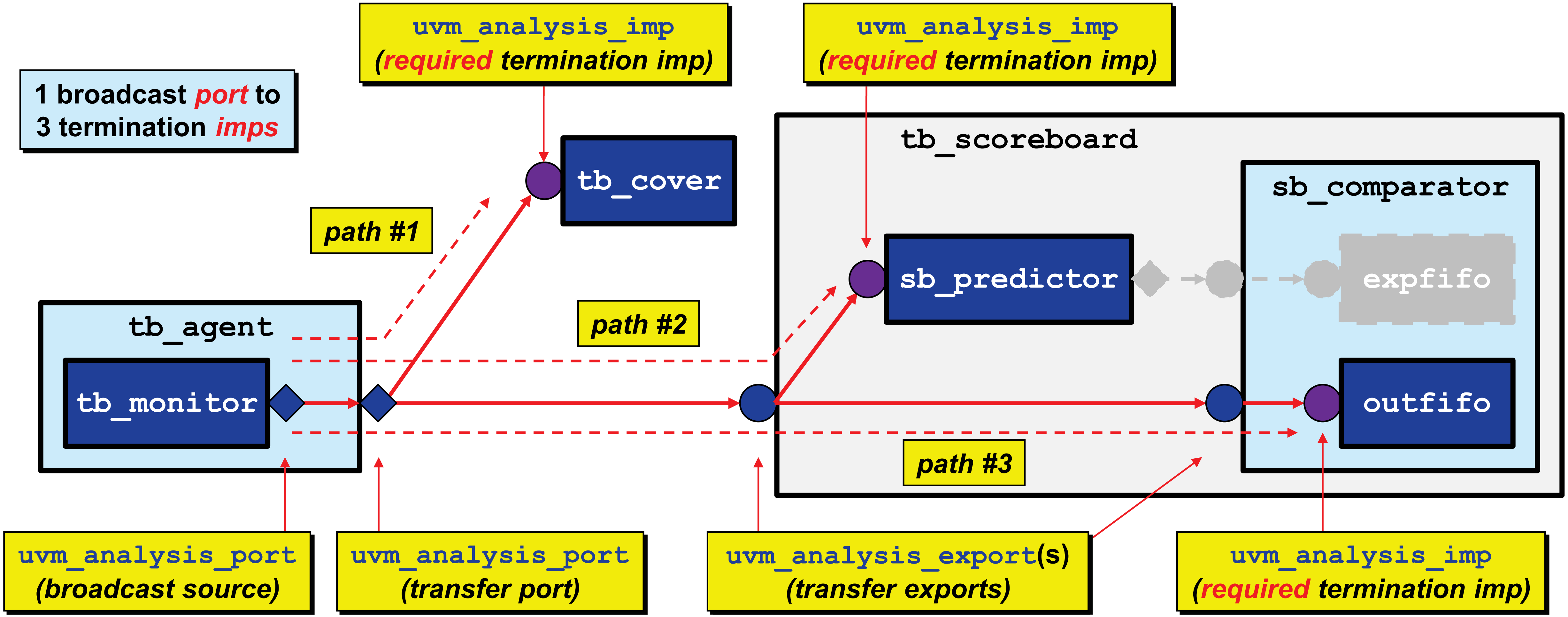
… the monitor becomes the broadcast source for the transaction

Analysis (broadcast) port

After the monitor samples signals from the DUT …

UVM Testbench Analysis Port Paths

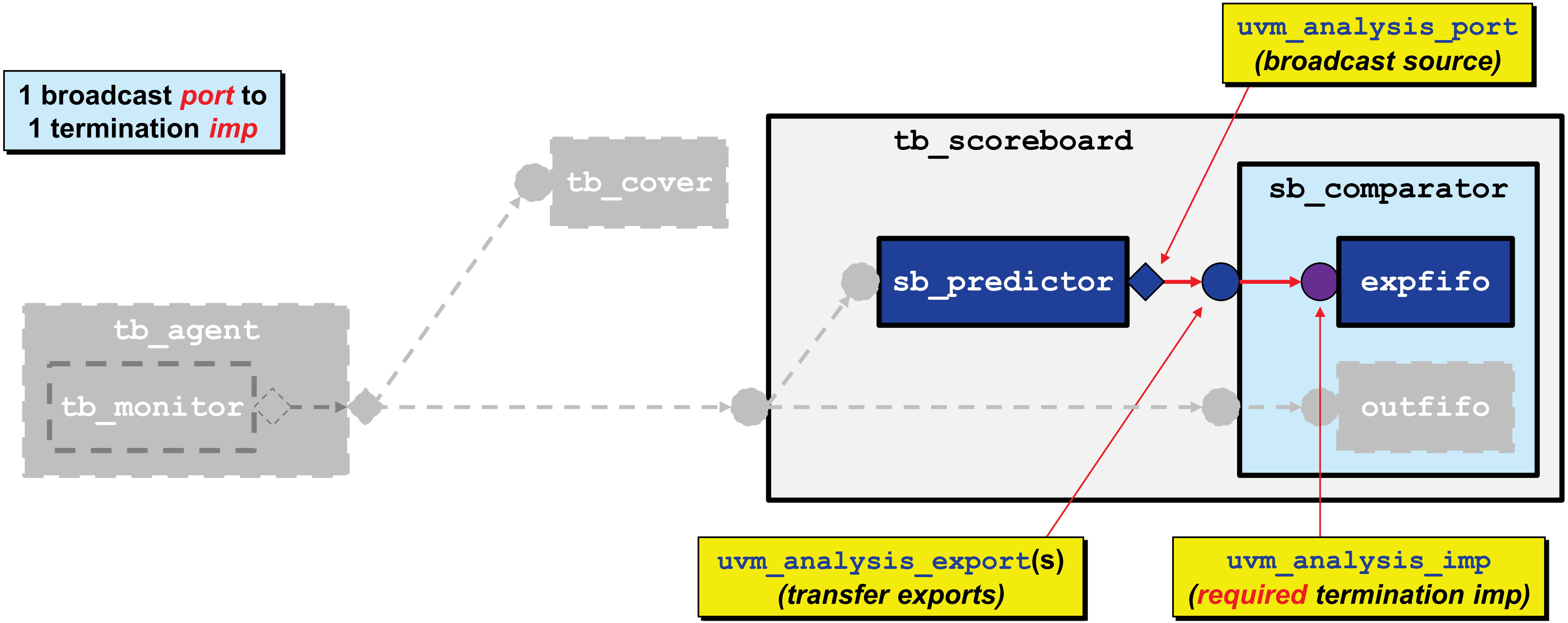
Common Paths - Monitor to Multiple Subscribers



UVM Testbench Analysis Port Paths

Common Paths - Predictor to Expected Transaction FIFO

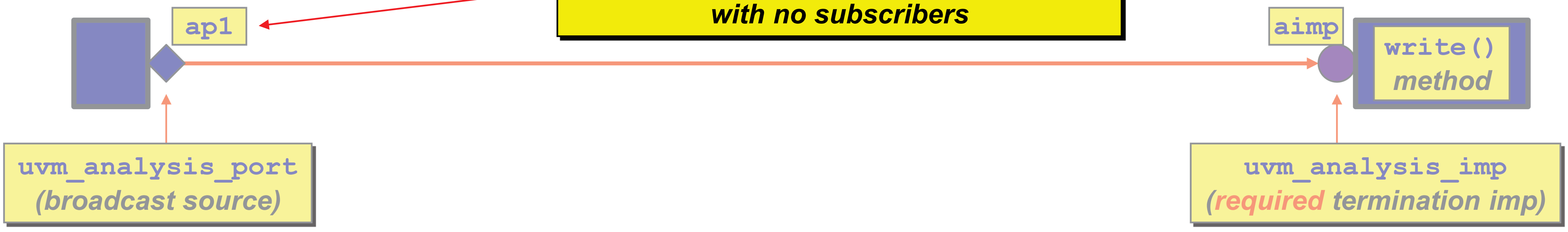
1 broadcast *port* to
1 termination *imp*

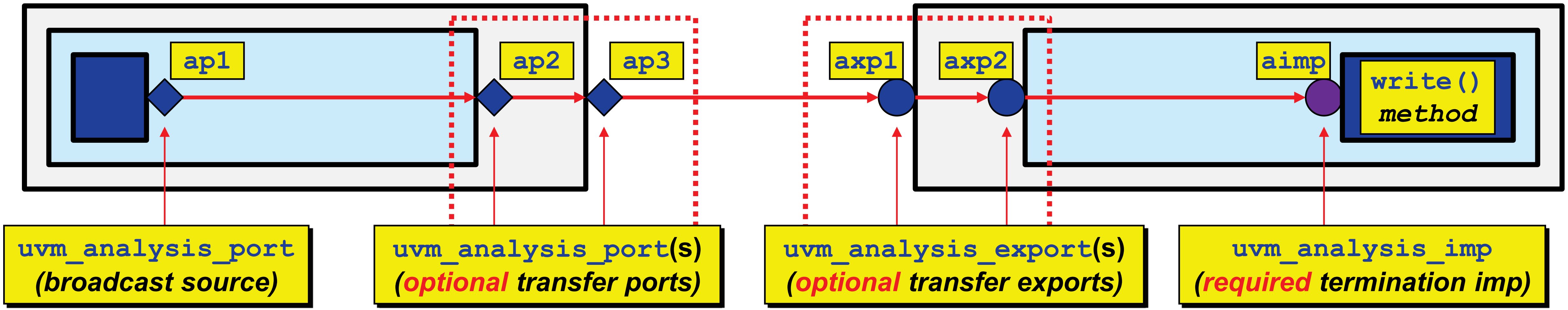


UVM Analysis Port Paths

LEGAL Paths

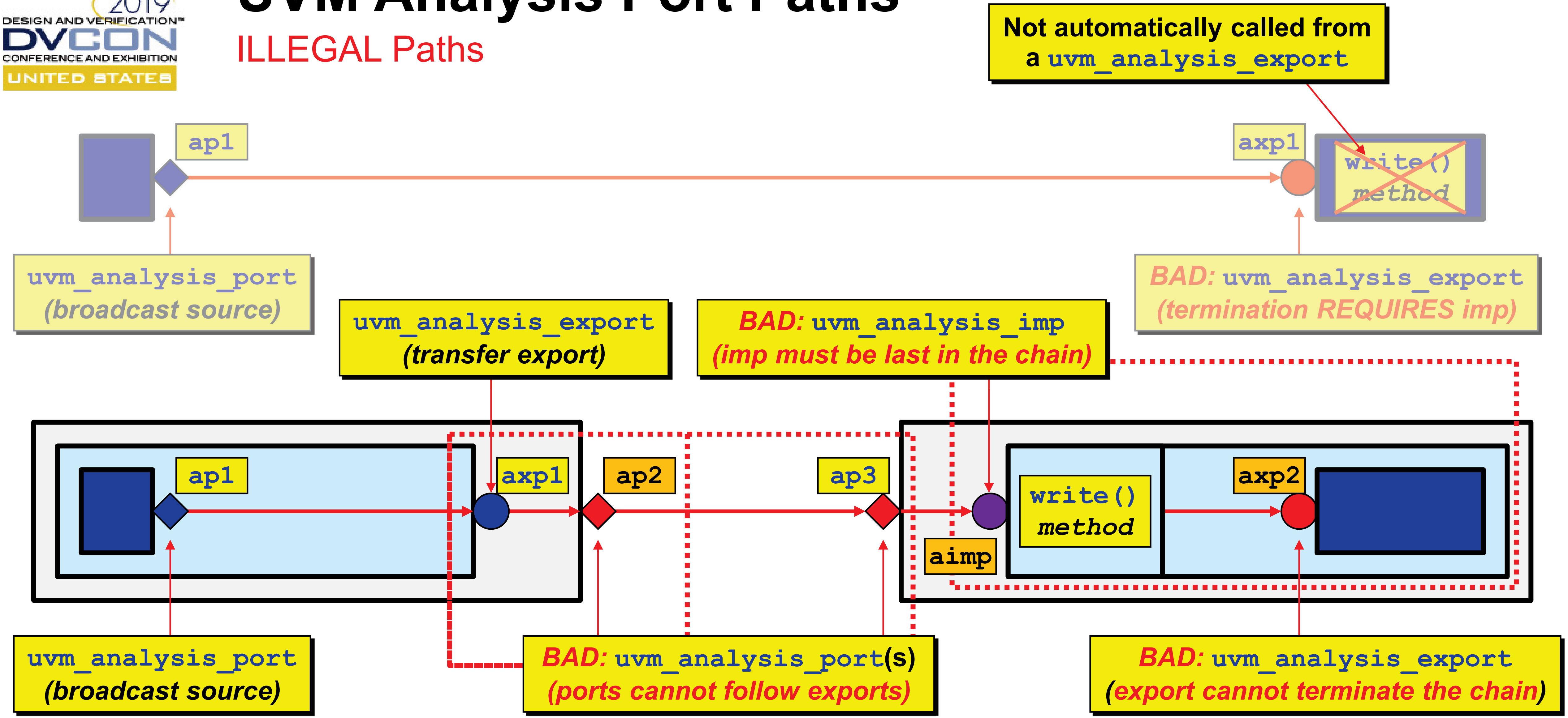
The most simple analysis path is a `uvm_analysis_port` *(broadcast source) with no subscribers*





UVM Analysis Port Paths

ILLEGAL Paths



Not automatically called from a `uvm_analysis_export`

`uvm_analysis_port` *(broadcast source)*

BAD: `uvm_analysis_export` *(termination REQUIRES imp)*

`uvm_analysis_export` *(transfer export)*

BAD: `uvm_analysis_imp` *(imp must be last in the chain)*

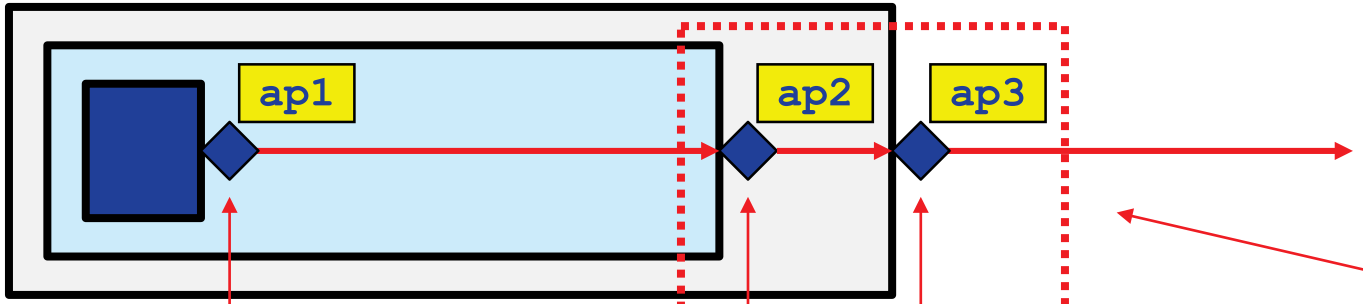
`uvm_analysis_port` *(broadcast source)*

BAD: `uvm_analysis_port`(s) *(ports cannot follow exports)*

BAD: `uvm_analysis_export` *(export cannot terminate the chain)*

UVM Analysis Ports

Recommended Usage



Outgoing transactions

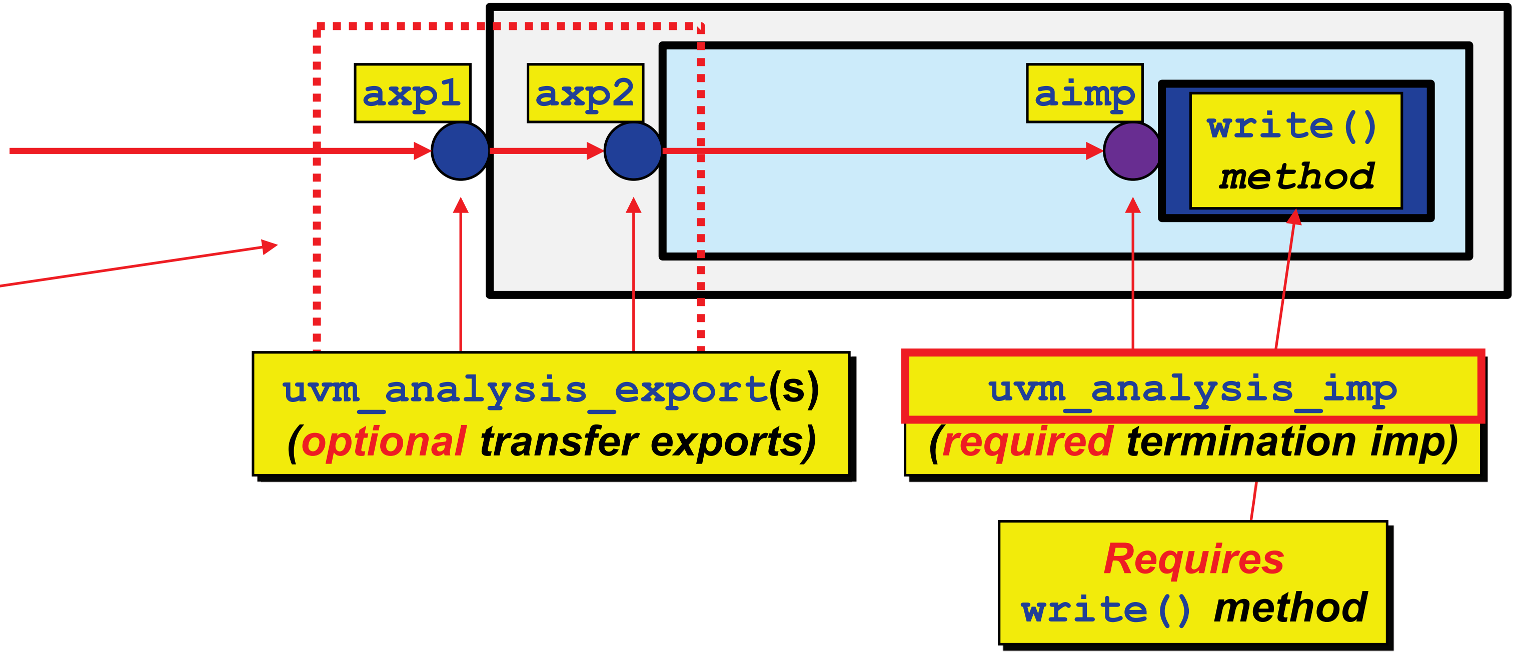
Use uvm_analsys_port(s)

uvm_analysis_port *(broadcast source)*

uvm_analysis_port (s) *(optional transfer ports)*

UVM Analysis Exports & Imps

Recommended Usage



Incoming transactions

Use uvm_analsys_export(s) & uvm_analysis_imp

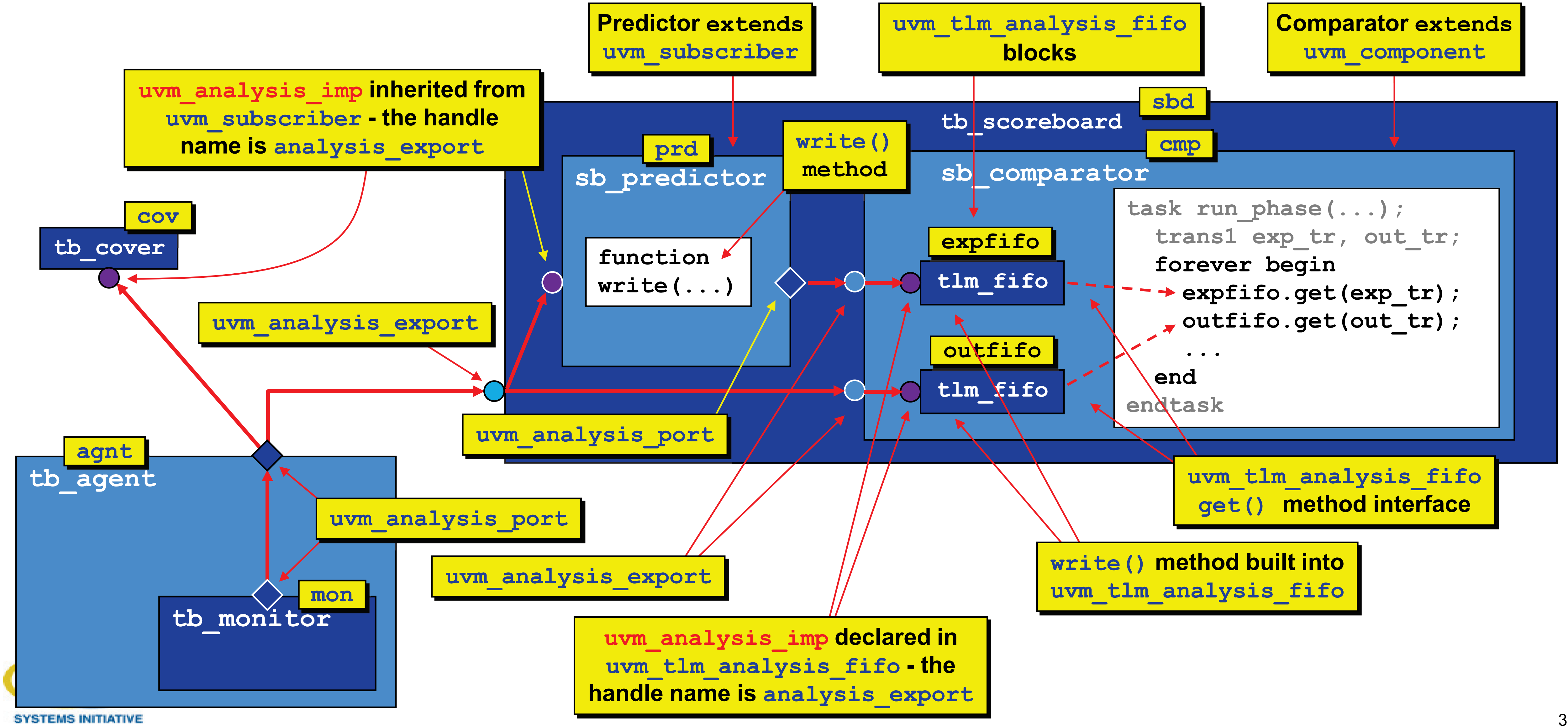
uvm_analysis_export(s) (*optional* *transfer exports*)

uvm_analysis_imp (*required* *termination imp*)

Requires write() *method*

Common Analysis Port Connections

Recommended Connections



Predictor extends uvm_subscriber

uvm_tlm_analysis_fifo blocks

Comparator extends uvm_component

uvm_analysis_imp inherited from uvm_subscriber - the handle name is analysis_export

write() method

uvm_analysis_export

uvm_analysis_port

uvm_analysis_port

uvm_analysis_export

uvm_tlm_analysis_fifo get() method interface

write() method built into uvm_tlm_analysis_fifo

uvm_analysis_imp declared in uvm_tlm_analysis_fifo - the handle name is analysis_export

```
function
write(...)
```

```
task run_phase(...);
  trans1 exp_tr, out_tr;
  forever begin
    expfifo.get(exp_tr);
    outfifo.get(out_tr);
    ...
  end
endtask
```

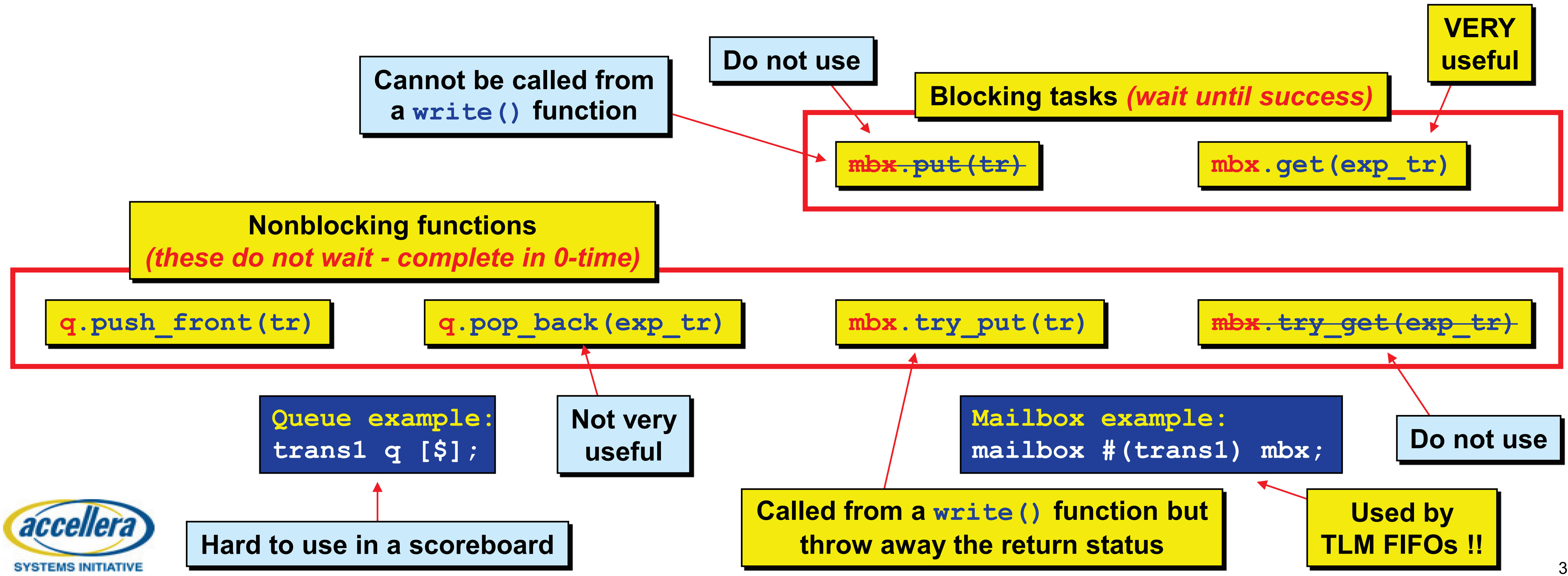
TLM FIFOs - Definitions & Usage

TLM FIFOs & Scoreboards

SystemVerilog Queues & Mailboxes

- Scoreboards typically store *expected* and *actual* transactions
- SystemVerilog has *queues* and *mailboxes*

Which should be used?



Blocking tasks *(wait until success)*

~~`mbx.put(tr)`~~

`mbx.get(exp_tr)`

Do not use

VERY useful

Cannot be called from a `write()` function

Nonblocking functions *(these do not wait - complete in 0-time)*

`q.push_front(tr)`

`q.pop_back(exp_tr)`

`mbx.try_put(tr)`

~~`mbx.try_get(exp_tr)`~~

```
Queue example:
trans1 q [$];
```

Not very useful

Hard to use in a scoreboard

```
Mailbox example:
mailbox #(trans1) mbx;
```

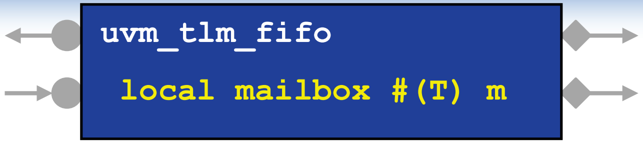
Do not use

Called from a `write()` function but throw away the return status

Used by TLM FIFOs !!

uvm_tlm_fifo

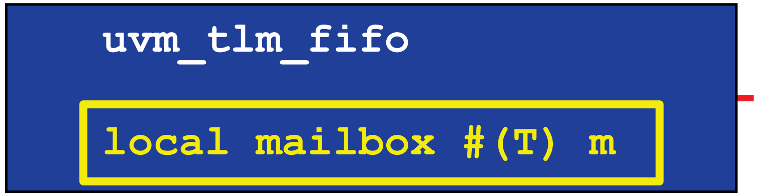
Most Common Usage



Although there are 2 *non-analysis* imp ports and 2 analysis ports on the uvm_tlm_fifo, they typically are not used

The uvm_tlm_fifo, will be constructed to be unbounded. Example:
`exp_fifo=new("exp_fifo", this, 0);`

0 means unbounded



try_put() method stores into the mailbox

get() method retrieves from the mailbox

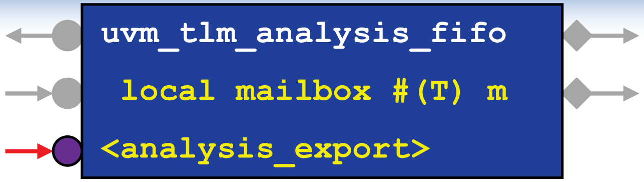
The uvm_analysis_imp write(tr) method will call void'(try_put(tr))

The scoreboard comparator will call the blocking get(tr) method and wait to retrieve a uvm_tlm_fifo transaction

void-cast to throw away the try_put() return-status
(*try_put() always succeeds on unbounded fifo's mailbox)*

uvm_tlm_analysis_fifo

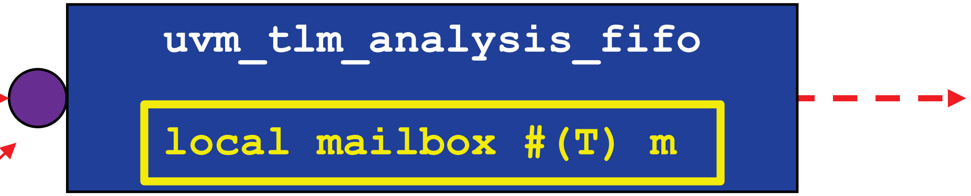
Most Common Usage



Although there are 2 *non-analysis* imp ports and 2 analysis ports on the uvm_tlm_fifo, they typically are not used

The uvm_tlm_analysis_fifo is unbounded by default

Termination of an analysis-path



get() method retrieves from the mailbox

uvm_analysis_imp with handle name analysis_export is almost always used

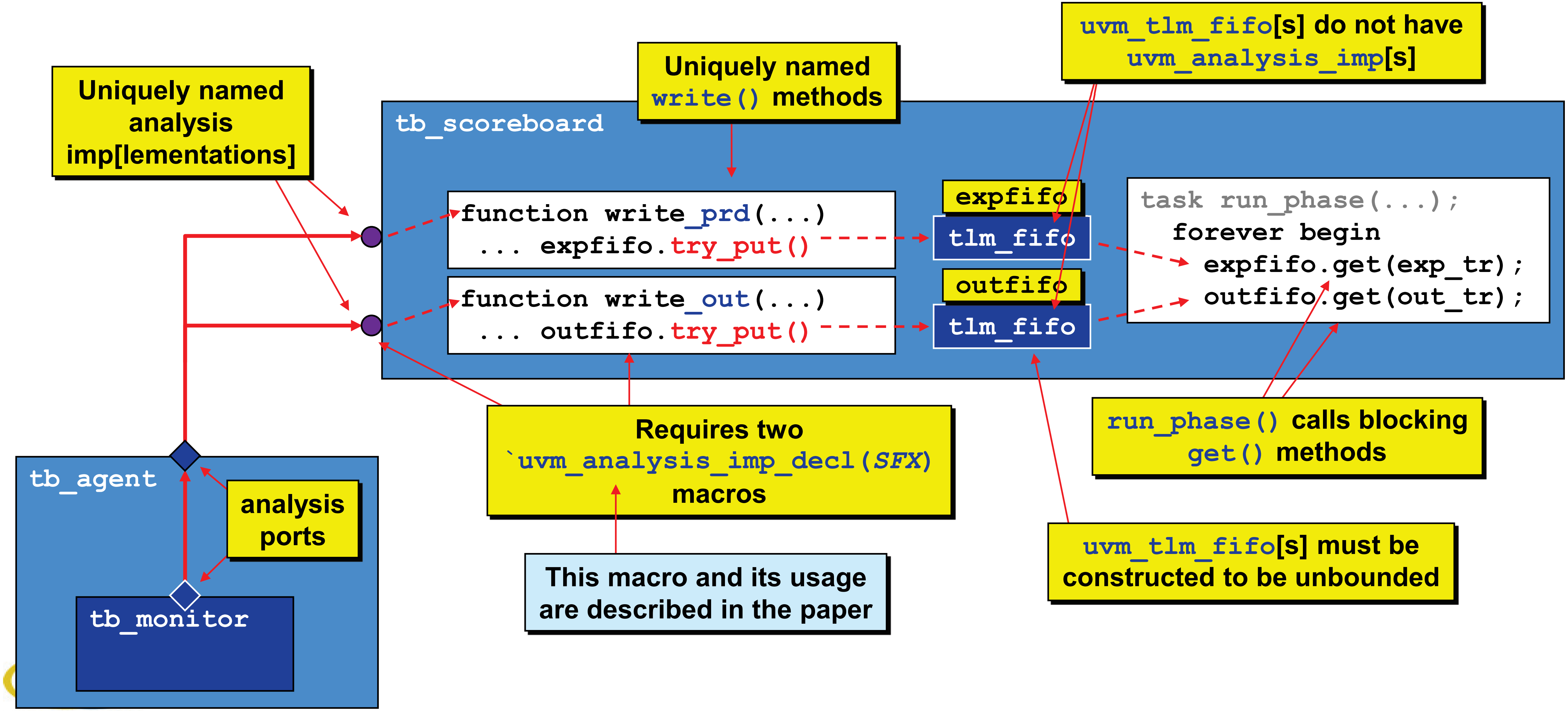
The scoreboard comparator still calls blocking get(tr) method and waits to retrieve a uvm_tlm_analysis_fifo transaction

imp already has write(t) method built-in

Internally executes:
`void'(this.try_put(t));`

Typical Scoreboard

Using uvm_tlm_fifos



tb_scoreboard

```
function write_prd(...)
  ... expfifo.try_put()
```

```
function write_out(...)
  ... outfifo.try_put()
```

expfifo tlm_fifo

outfifo tlm_fifo

```
task run_phase(...);
  forever begin
    expfifo.get(exp_tr);
    outfifo.get(out_tr);
```

tb_agent

tb_monitor

Uniquely named analysis imp[lementations]

Uniquely named write() methods

uvm_tlm_fifo[s] do not have uvm_analysis_imp[s]

analysis ports

Requires two `uvm_analysis_imp_decl(*SFX*) macros

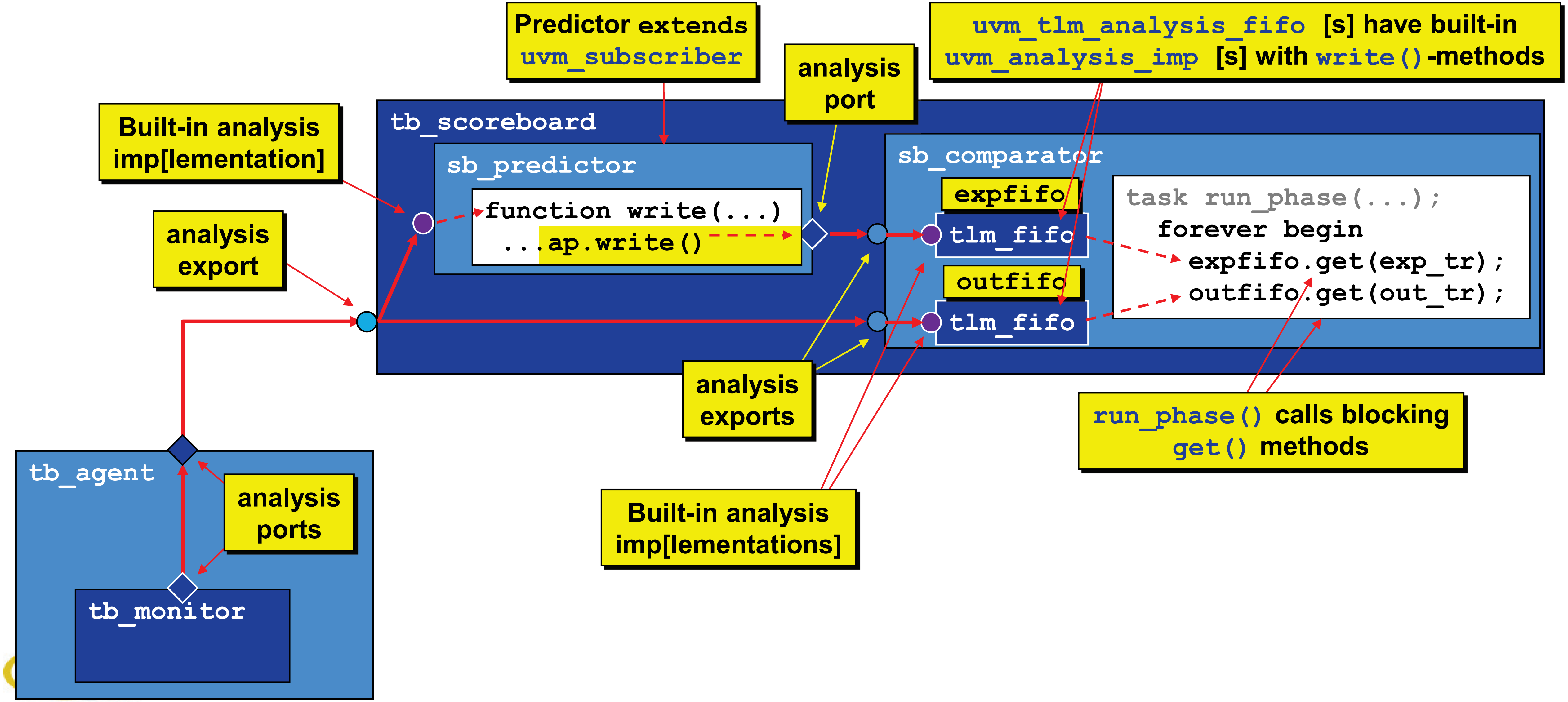
This macro and its usage are described in the paper

run_phase() calls blocking get() methods

uvm_tlm_fifo[s] must be constructed to be unbounded

Typical Scoreboard

Using uvm_tlm_analysis_fifos



Predictor extends `uvm_subscriber`

analysis port

`uvm_tlm_analysis_fifo` [s] have built-in `uvm_analysis_imp` [s] with `write()` -methods

Built-in analysis imp[lementation]

analysis export

analysis exports

Built-in analysis imp[lementations]

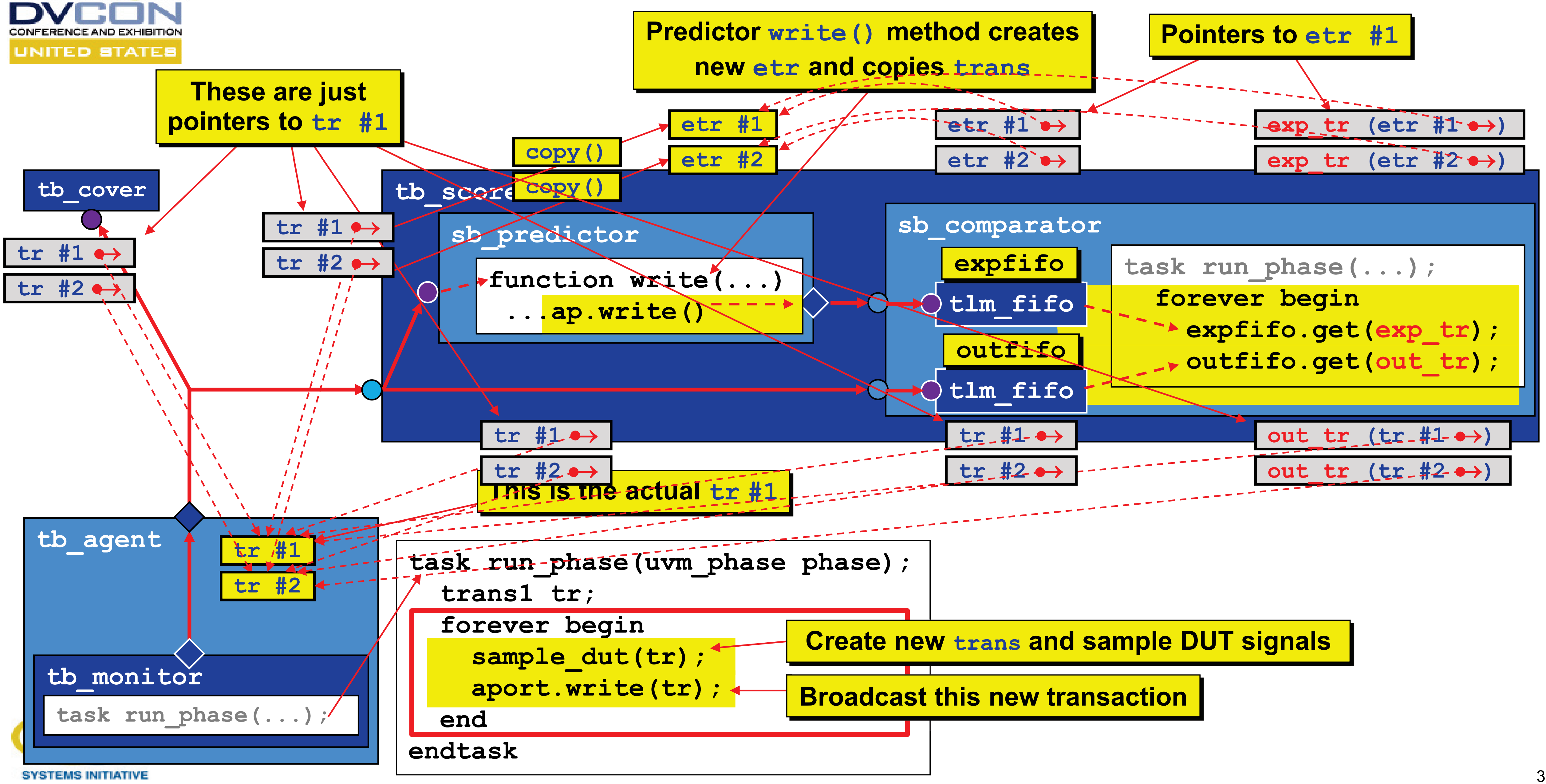
`run_phase()` calls blocking `get()` methods

analysis ports

```
function write(...)
  ...ap.write()
```

```
task run_phase(...);
  forever begin
    expfifo.get(exp_tr);
    outfifo.get(out_tr);
```

Creating & Copying Transactions



```
task run_phase(...);
  forever begin
    expfifo.get(exp_tr);
    outfifo.get(out_tr);
```

These are just pointers to tr #1

Predictor write() method creates new etr and copies trans

Pointers to etr #1

This is the actual tr #1

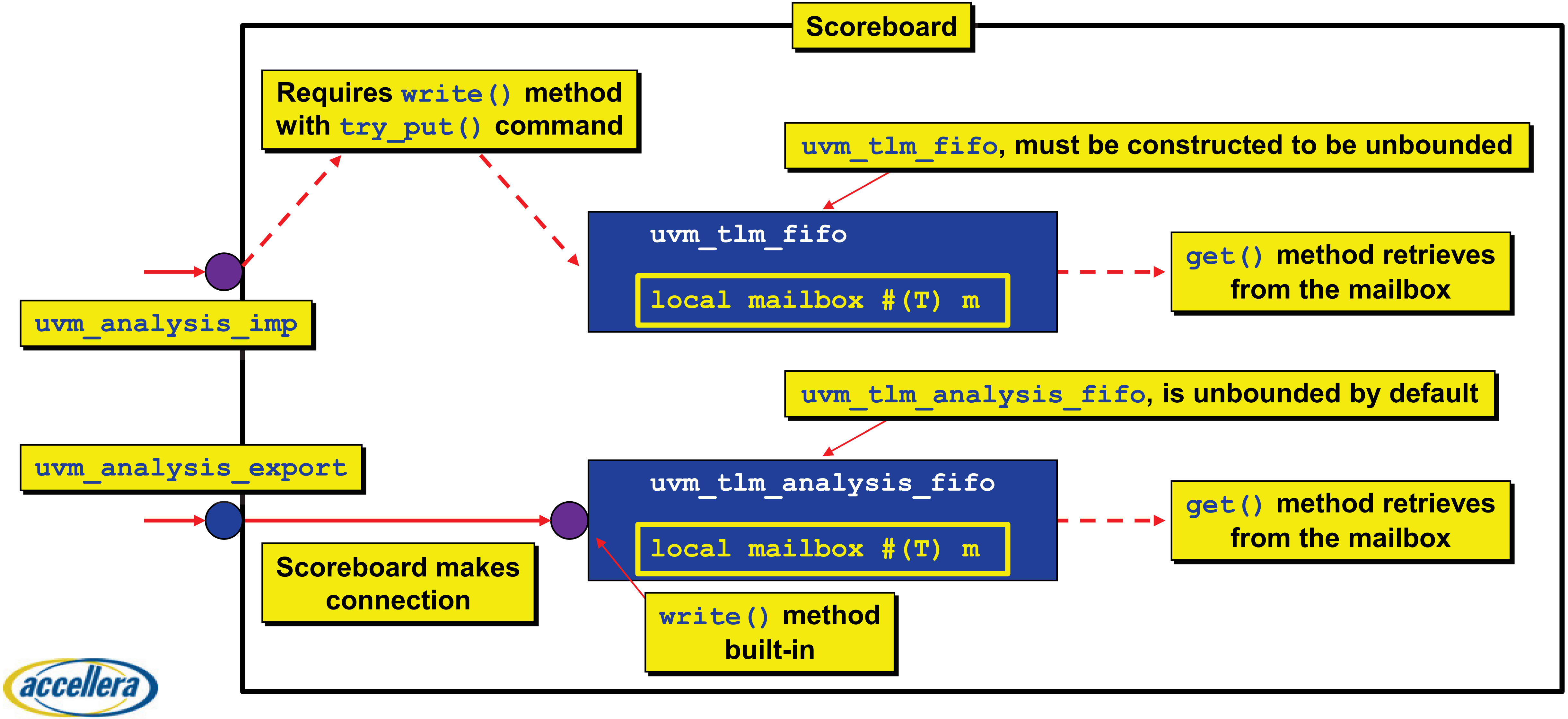
```
task run_phase(uvm_phase phase);
  trans1 tr;
  forever begin
    sample_dut(tr);
    aport.write(tr);
  end
endtask
```

Create new trans and sample DUT signals

Broadcast this new transaction

Comparing TLM FIFOs

uvm_tlm_fifo -vs- uvm_tlm_analysis_fifo


Scoreboard
Requires write() method with try_put() command
uvm_tlm_fifo, must be constructed to be unbounded
uvm_tlm_fifo
local mailbox #(T) m
get() method retrieves from the mailbox
uvm_analysis_imp
uvm_tlm_analysis_fifo, is unbounded by default
uvm_analysis_export
uvm_tlm_analysis_fifo
local mailbox #(T) m
get() method retrieves from the mailbox
Scoreboard makes connection
write() method built-in

Ports & Exports

Is the Naming Backwards?

- How to think about *Ports* and *Exports*
- Automobile features:
  - Steering wheel
  - Accelerator pedal
  - Brake pedal
  - Hands-free Bluetooth-phone connection

Analysis Path Basics

In the software world, this is known as the *"Observer Pattern"*

How do analysis port-paths work?

How Does UVM Work?

On previous slides

- We have learned about analysis ports & TLM FIFOs

You now know how to use:

```
uvm_analysis_port
uvm_analysis_export
uvm_analysis_imp
uvm_tlm_fifo
uvm_tlm_analysis_fifo
```

- You do not have to know how UVM works

You now have enough knowledge to use analysis components

- The best engineers want to have *some* understanding on how UVM works
- The remaining slides show how UVM makes subscribers work

This is *NOT* UVM code !!

This is a *basic* version of what UVM does internally

These slides show how UVM uses *queues* and `foreach` loops to call each subscriber's `write()` method

This is a high-level tutorial on how monitors and subscribers work

This is not exactly how UVM works, *but it is close*

Monitor with Multiple Subscribers

Goal

- Create a *Monitor* that can connect to any number of subscribers and can call a `write()` method from each subscriber ***without modifying the Monitor code***

– **Version #1** ← **top module must know subscriber handle names in the Monitor** ✘

The monitor …

Must declare each subscriber handle ✘

Has no `connect()` method Must copy handles by name ✘

Must call `write()` method for each subscriber ✘

– **Version #2** ← **Monitor w/ generic `connect()` method to hide subscriber handle names** ✓

UVM Like!

The monitor …

Has queue of subscriber handles ✓

Defines common `connect()` method for all subscribers ✓

Uses `foreach` loop to call `write()` methods using queued subscriber handles ✓

Monitor & Subsc

Version 1 - No connect() method

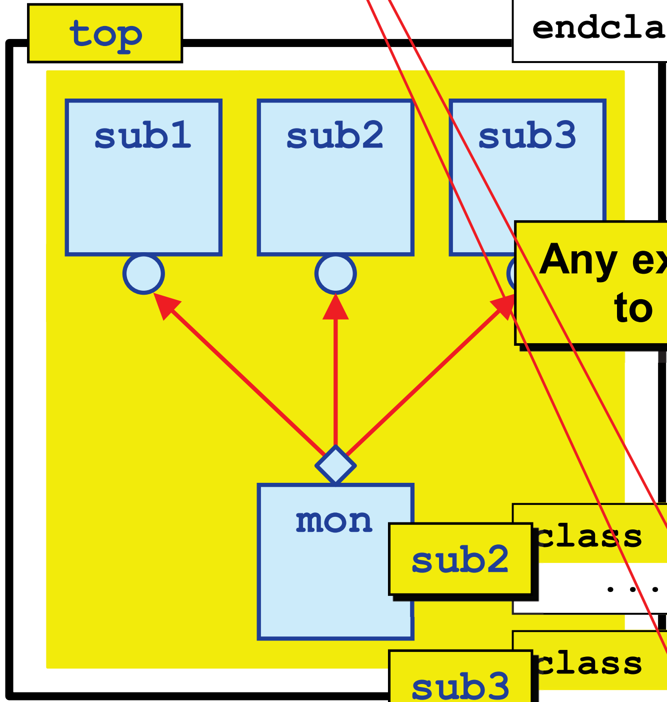
virtual analysis_if base class

Extended classes must implement write() method

```
virtual class analysis_if;
  pure virtual task write(trans1 t);
endclass
```

```
class subscriber1 extends analysis_if;
  virtual task write(trans1 t);
    $display("subscriber1: ",
       "received ...", ...);
  endtask
endclass
```

sub1



Each subscriber handle is copied to the ap1-3 handles in monitor1

Any extended ... copied
to a base c ... ng)

```
...
mon.ap1 = sub1;
mon.ap2 = sub2;
mon.ap3 = sub3;
...
```

In top module

sub2

```
class subscriber2 extends analysis_if;
   ... virtual task write(...) ...
```

sub3

```
class subscriber3 extends analysis_if;
   ... virtual task write(...) ...
```

```
class monitor1;
  analysis_if ap1;
  analysis_if ap2;
  analysis_if ap3;

  task run();
    trans1 t = new();
    repeat(5) begin
      void'(t.randomize());
      $display("monitor:   ",
       "**BROADCAST** ...", ...);
      ap1.write(t);
      ap2.write(t);
      ap3.write(t);
    end
  endtask
endclass
```

subscriber1 sub1 to ap1

subscriber2 sub2 to ap2

subscriber3 sub3 to ap3

Monitor & Subscribers

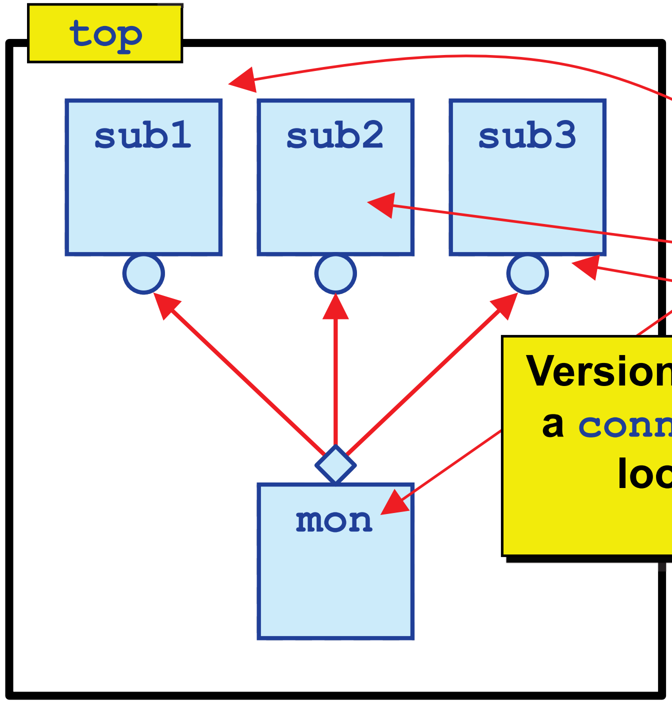
Version 1 - No connect() method

Declare monitor1 and subscriber1-3 handles

new()-construct mon and sub1-3

Copy sub1-3 handles to ap1-3 handles in monitor1

Call the mon.run() task



```
module top;
  import tb_pkg::*;

  monitor1    mon;
  subscriber1 sub1;
  subscriber2 sub2;
  subscriber3 sub3;

  initial begin
    mon  = new();
    sub1 = new();
    sub2 = new();
    sub3 = new();

  end
endmodule
```

Version 2 will add an analysis_if queue, a connect() method and use a foreach loop to call the write() methods *(next slide)*

```
class monitor1;
  analysis_if ap1;
  analysis_if ap2;
  analysis_if ap3;

  task run();
    trans1 t = new();
    repeat(5) begin
      void'(t.randomize());
      $display("monitor:   ",
       "**BROADCAST** ...", ...);
      ap1.write(t);
      ap2.write(t);
      ap3.write(t);
    end
  endtask
endclass
```

Monitor must declare each analysis_if

With no connect() method in monitor1, the top module must reference names declared in monitor1

Repeat 5 times

randomize() transaction

Separately call each ap[#].write() method

Monitor & Subscribers

Version 2 - Adds analysis_if queue

Queue of analysis_if handles ✓

Common connect() method ✓

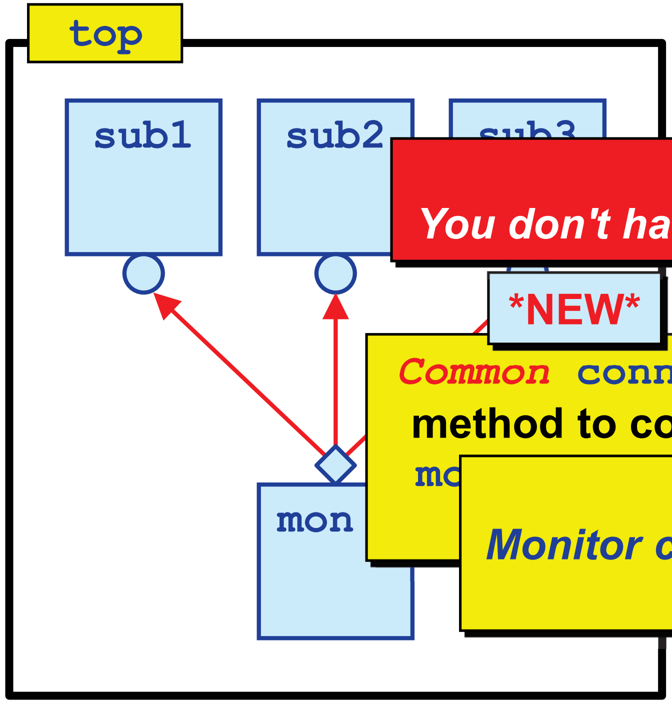
foreach calls write() methods ✓

No change from Version 1

```
module top;
  import tb_pkg::*;

  monitor2    mon;
  subscriber1 sub1;
  subscriber2 sub2;
  subscriber3 sub3;

  initial begin
```

This is what UVM does!
You don't have to do this in your UVM testbenches

```
    sub2 = new();
    sub3 = new();
    mon.connect(sub1);
    mon.connect(sub2);
```



NEW

Common connect() method to connect mo

Goal achieved!
Monitor code does not require modifications to add more subscribers!

```
class monitor2;
  analysis_if ap[$];

  function void connect (analysis_if port);
    ap.push_back(port);
  endfunction

  task run();
    trans1 t = new();
    repeat(5) begin
      void'(t.randomize());
      $display("monitor:   ",
               "**BROADCAST** ...", ...);
      foreach(ap[i]) ap[i].write(t);
    end
  endtask
endclass
```

NEW

Monitor declares queue of analysis_if ports

NEW

Each call to connect() method will push_back another analysis_if onto the ap-queue

Common connect() method

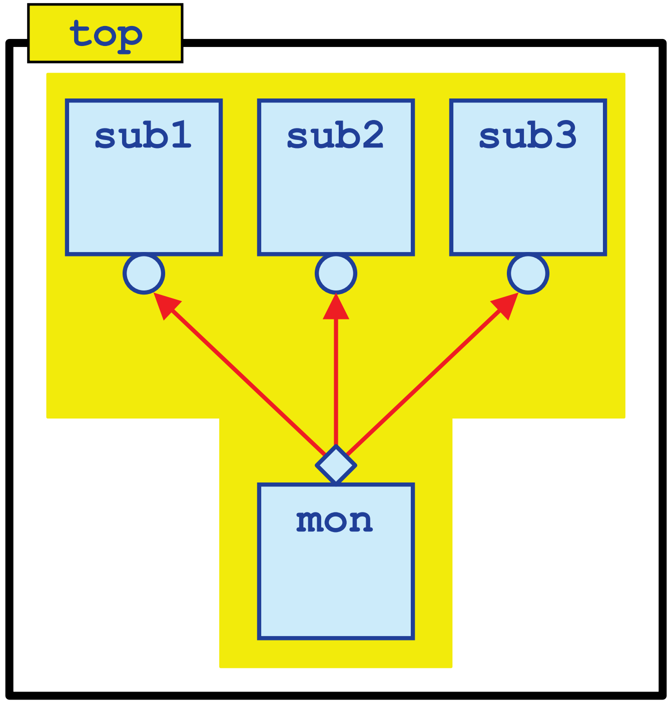
NEW

Each subscriber's write() method is called from the ap-queue

More subscribers could be added to top module without modifying monitor2 code

Monitor & Subscribers

Simulation Output



```
Randomized trans1 values addr=f9  data=50
monitor:    **BROADCAST** addr=f9  data=50
subscriber1: received     addr=f9  data=50
subscriber2: received     addr=f9  data=50
subscriber3: received     addr=f9  data=50

Randomized trans1 values addr=e9  data=27
monitor:    **BROADCAST** addr=e9  data=27
subscriber1: received     addr=e9  data=27
subscriber2: received     addr=e9  data=27
subscriber3: received     addr=e9  data=27

...
```

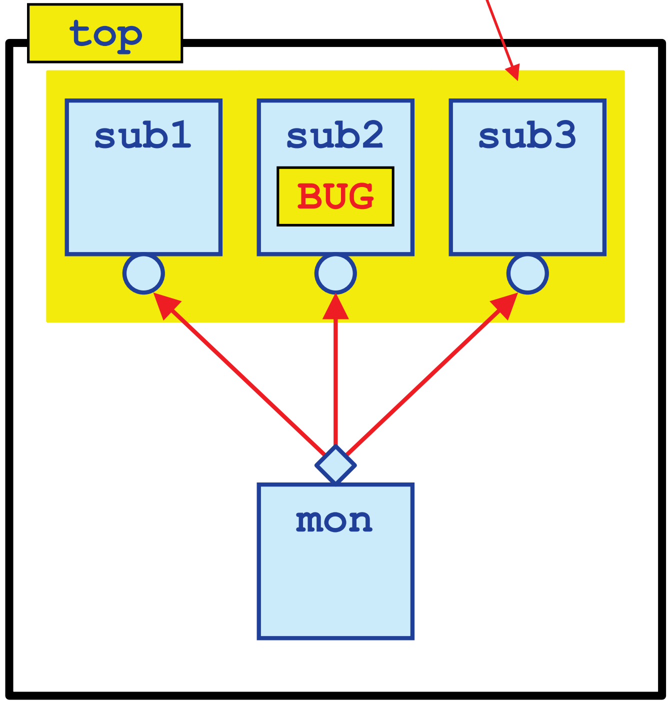
Each subscriber has seen the exact same `addr` and `data` values that were broadcast to all subscribers

Subscriber2 BUG

Version 3 - modifies transaction values

```
class subscriber1 extends analysis_if;
  virtual task write(trans1 t);
    $display("subscriber1: ", "received addr=%2h  data=%2h", t.addr, t.data);
  endtask
endclass
```

subscriber1 has the original transaction addr & data values



sub3 sees corrupted transaction

```
class subscriber2 extends analysis_if;
  virtual task write(trans1 t);
    $display("subscriber2: ", "received addr=%2h  data=%2h", t.addr, t.data);

`ifdef BUG
    t.addr = 8'hFF;
    t.data = 8'h00;
    $display("subscriber2: ", "set      addr=%2h  data=%2h", t.addr, t.data);
`endif

  endtask
endclass
```

BUG: subscriber2 modifies the addr & data of the broadcast transaction

NEVER **modify the broadcast transaction !!**

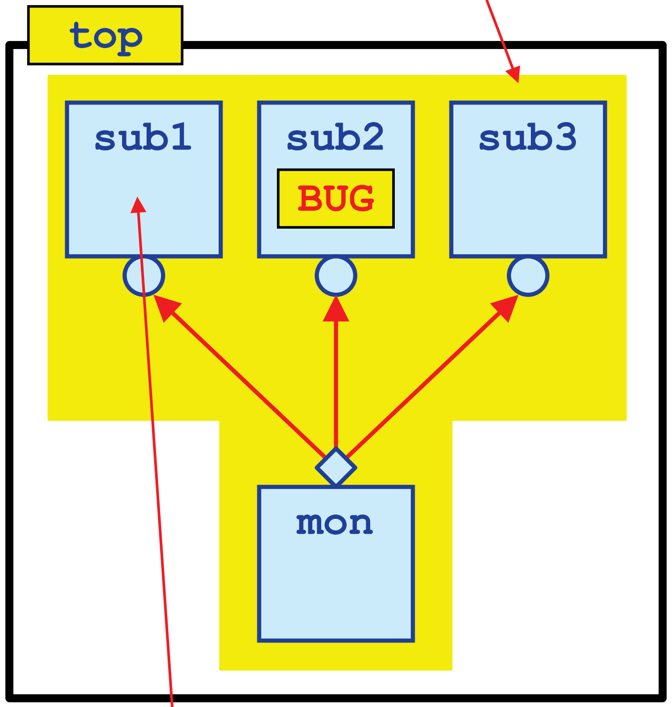
```
class subscriber3 extends analysis_if;
  virtual task write(trans1 t);
    $display("subscriber3: ", "received addr=%2h  data=%2h", t.addr, t.data);
  endtask
endclass
```

subscriber3 now sees the modified transaction addr & data values

Monitor & Subscribers

BUG: Simulation Output

sub3 sees corrupted transaction



Depending on how the subscribers are pushed onto the ap - queue, sub1 might also see the bug

```
Randomized trans1 values addr=f9   data=50
monitor:    **BROADCAST** addr=f9   data=50
subscriber1: received     addr=f9   data=50
subscriber2: received     addr=f9   data=50
subscriber2: set          addr=ff   data=00
subscriber3: received     addr=ff   data=00

Randomized trans1 values addr=e9   data=27
monitor:    **BROADCAST** addr=e9   data=27
subscriber1: received     addr=e9   data=27
subscriber2: received     addr=e9   data=27
subscriber2: set          addr=ff   data=00
subscriber3: received     addr=ff   data=00

...
```

Transaction Copy() Method

- All subscribers receive a handle to the *same* broadcast transaction
- A subscriber should ***NEVER*** modify contents of the received transaction
- Any subscriber that modifies transaction contents ***MUST take a copy before making modifications***

Summary & Conclusions

- Analysis ports are ports that broadcast transactions to 0 or more destinations
- Each subscriber chain terminates with a `uvm_analysis_imp` and corresponding `write()` method
- Subscribers should ***NEVER*** modify the broadcast transaction
- Subscribers need to use the transaction in 0-time

  -OR-
- Subscribers need to take a local copy
- If a component has multiple `imp`-inputs, use the macro:

  `` `uvm_analysis_imp_decl(SFX) ``

  This is described in the paper
- The `uvm_tlm_analysis_fifo` has a built-in `uvm_analysis_imp` port

  Great feature for terminating an analysis path in a scoreboard
- Prove that the scoreboard analysis paths are working

DO NOT ASSUME **that the analysis paths are working correctly !!**

Resources Summary

- Go go Accellera website

  www.accellera.org

Many great resources on this web site

- Register for free access to the DVCon 2017 and DVCon 2018 videos

To watch these presentations, go to:
videos.accellera.org/videos.html

- forums.accellera.org/

Access the SystemVerilog and UVM Forums

- Get a free IEEE login

Linked from
www.accellera.org/downloads/ieee

1800.2-2017 - IEEE UVM

- https://ieeexplore.ieee.org/document/7932212

1800-2017 - IEEE SystemVerilog

Downloading PDF documents requires IEEE login
(You can create a free IEEE login account)

- https://ieeexplore.ieee.org/document/8299595

Reference Material

DVCon 2018 Tutorial: IEEE-Compatible UVM Reference Implementation and Verification Components

DVCon 2017 Tutorial: Introducing IEEE 1800.2 - The Next Step for UVM

To watch these presentations, go to:
`videos.accellera.org/videos.html`

DVCon 2017 - UVM Features Described

Thomas Alsop - Intel Corp.

Reference Material

Slide #	
14 -	Introduction to IEEE and Backward Compatibility
15 -	BCL compliance to the IEEE 1800.2 spec
16 -	Implementations artifacts and additive but non-IEEE APIs
17 -	Deprecation policy and roadmap
18 -	Removal of pre-1.2 deprecated code - Motion pending
19 -	APIs that changed from 1.2 to IEEE - Motion pending

DVCon 2017 - UVM Features Described

Srivatsa Vasudevan - Synopsys, Inc.

Reference Material

Slide #	
28 -	UVM Policy Classes - **`copy`**, **`compare`**, **`print`**, **`pack`**, **`record`** all have policy classes
29 -	**`uvm_policy`** - users can apply different printer or compare policy + many accessor methods
30 -	**`uvm_packer`** - new pack / unpack capabilities
31-32 -	**`uvm_copier`** - signature of **`copy()`** has changed to allow **`uvm_copier`**
33-34 -	**`uvm_comparer`** - provides new accessor methods
35-36 -	**`uvm_printer`** - new printer knobs & accessor methods
37-39 -	**`uvm_line_printer`** / **`uvm_table_printer`** / **`uvm_tree_printer`**
40 -	**`uvm_recorder`** - new methods
41 -	Summary of core utility policies

DVCon 2017 - UVM Features Described

Srivatsa Vasudevan - Synopsys, Inc.

Reference Material

Slide #

43-45 - UVM factory now supports abstract objects (`virtual` classes)

47 - `uvm_component` - can turn off `apply_config_settings()`

49 - `uvm_object` - small modifications & new methods

50 - minor `uvm_transaction` modifications

51 - *Removed* from IEEE 1800.2 - *Deemed as not standard worthy*

- `uvm_comparator`
- `uvm_algorithmic_comparator`
- `uvm_in_order_comparator`

53-54 - `uvm_report_object` - minor modifications

55 - `uvm_report_server` - `UVM_FILE` type change

56 - `uvm_report_catcher` - minor modifictions

58 - Callbacks now extend from `uvm_callback` - functions documented

DVCon 2017 - UVM Features Described

Mark Glasser - NVIDIA Corporation

Reference Material

Slide #	
63 -	Summary of TLM Mantis Items
68 -	Register models - documentation enhanced / system level / dynamic
69 -	Reg model unlock - models can now be unlocked & re-locked
70 -	Register changes - `virtual` and non-`virtual` classes

DVCon 2017 - UVM Features Described

Srinivasan Venkataramanan - CVC Pvt., Ltd.

Reference Material

Slide #	
76 -	Details regarding Typical UVM Architecture
77 -	Description of UVM Mechanics
81-105 -	Description of VerifWorks Go2UVM package and capabilities

DVCon 2018 - UVM Features Described

Justin Refice - Nvidia

Reference Material

Slide #	
3-7 -	Accellera & IEEE UVM responsibilities
8 -	Transitioning from UVM 1.2 to IEEE 1800.2 UVM
8 -	`` `UVM_ENABLE_DEPRECATED_API `` to keep using UVM 1.2
9-12 -	Deprecation notes and transitioning considerations
13 -	Recommended Steps of Updating to IEEE 1800.2

DVCon 2018 - UVM Features Described

Reference Material

Mark Strickland - Cisco Systems Mark Peryer - Mentor, a Siemens Busin

Slide #	
17 -	`uvm_object` - New UVM seeding / new methods for configuration and policies
18 -	`do_execute_op` - call-back to add flexibility in field operations
19 -	Configuration considerations - field macros execute `do_execute_op`
21 -	UVM Policy Classes - `copy`, `compare`, `print`, `pack`, `record` all have policy classes that extend from `uvm_policy`
22 -	Policy extensions and methods
23 -	`do_method()` use model changes
24 -	Standard method changes: `compare()` *calls* `do_execute_op()` *calls* `do_compare()`
26-28 -	`copy()` / `do_copy()` / `copy_object()` / `uvm_copier` example
29-31 -	`record()` / `do_record()` / `detail_extension` / `uvm_recorder` example

DVCon 2018 - UVM Features Described

Reference Material

Mark Strickland - Cisco Systems Mark Peryer - Mentor, a Siemens Business

Slide #	
32 -	Scoreboards need to compare objects of differing types
33-35 -	`compare()` / `do_compare()` / `uvm_comparer` / `do_execute_op()` with scoreboard example
36 -	`pack()` / `unpack()` - small enhancements
37-	UVM printer policies now use `uvm_printer_element` & `uvm_printer_element_proxy`
38-43 -	JSON printer example with details

DVCon 2018 - UVM Features Described

Uwe Simm - Cadence Design Systems

Reference Material

Slide #

45 - UVM abstract factory - can now register and override `virtual` classes

46-50 - Abstract UVM factory examples

51 - Pre-IEEE 1800.2 UVM initialization

52 - New IEEE 1800.2 reliable UVM initialization - describes `uvm_coresevice_t ::get`() / `uvm_init()` / `run_test()`

53-56 - UVM deferred initialization examples

57-58 - `uvm_run_test_callback` / `pre_run_test()` / `post_run_test()` / `pre_abort()`

59-62 - `uvm_reg_block.lock_model()` / `unlock_model()`

63 - Miscellaneous `uvm_reg` notes & changes including `uvm_door_e`

DVCon 2018 - UVM Features Described

Srivatsa Vasudevan - Synopsys

Reference Material

Slide #

65-66 - `apply_config_settings()` for `` `uvm_field_* `` macros user controllable

67-68 - `set_local()` replaces `set_*_local()` methods

69-71 - Callbacks now extend from `uvm_callback` - users can call `all_callbacks[$]`

72-74 - Report severity is now `UVM_NONE` for `uvm_report_error`

76 - `` `uvm_do `` replaces all earlier `` `uvm_do_* `` macros

77 - `` `uvm_do_* `` deprecation notes

Thank you!