World Class Verilog, SystemVerilog & OVM/UVM Training

UVM Transactions - Definitions, Methods and Usage

Clifford E. Cummings
Sunburst Design, Inc.
cliffc@sunburst-design.com
www.sunburst-design.com

ABSTRACT

Fundamental questions most novice UVM users have include: Why uses classes instead of structs to define transactions for verification environments? What are advantages of using classes to represent transactions in a verification environment? What methods should be defined in a UVM transaction class and why are there both field macros and do_methods() for creating the transaction methods?

This paper will detail advantages related to using class-based transactions and answer questions about why there is so much confusion surrounding transaction method definitions and usage. This paper will also detail transaction method usage and field definition guidelines and tradeoffs.

Table of Contents

Table of Tables

Table of Figures

Table of Examples

1. Introduction

All advanced class-based verification methodologies use classes to represent transactions, but why? Why not use structs?

To advanced users the answers are obvious but to novice users the questions never seem to be addressed in any literature. The problem is, most existing UVM texts and reference guides were written by really, really smart software engineers that assume that all users naturally know the answer to this and many other questions, which is not a valid assumption.

The first step to understand the answers to these questions is to compare class-based transaction capabilities to struct-based transaction capabilities.

This paper will also go into detail on the creation of transaction classes with standard transaction methods. The methods will be created using two techniques, (1) `do_methods()` and (2) UVM field macros.

2. Classes -vs- structs

New users often ask the question, why use class types instead of structs for verification?

To better understand why classes are used instead of structs, it is useful to compare the different capabilities between classes and structs in SystemVerilog.

- Classes and structs both have multiple fields.
- Classes can have randomized fields while struct fields cannot be automatically randomized.
- Classes can include randomization constraints while structs cannot include automatic randomization constraints.
- Classes can have important built-in methods while structs cannot have built-in methods.
- Classes are a dynamic type and you can generate as many as you need at runtime while structs are a static type and the user must anticipate and statically declare all required structs at the beginning of the simulation.
- Class types can be extended while new versions of a struct must be copied from the original version and new fields added.
- Classes can be put into a UVM factory for easy runtime substitution while structs cannot.

Classes are basically dynamic, ultra-flexible structs that can be easily randomized, easily control the randomization, and be created whenever they are needed. Classes have the multiple field encapsulation capability that exist in structs, plus so much more. That is why classes are the preferred structure to represent testbench transactions.

Another advantage shared by both classes and structs is that they are passed around the testbench as a unit, whether there is one signal or 1,000 signals in the transaction, so it is easy to pass signals around the testbench environment with single unit operations. If signals are added or removed from the transaction, most of the testbench structure requires no modification. There are

just a few testbench components that need to interact with all of the component signals individually. Some of those components will be discussed in later sections.

3. Transaction class types

Once it is accepted that transactions should be class types, the next question is what should UVM transaction classes be? UVM testbench transactions are all extensions of the **`uvm_sequence_item`** type, which is a derivative of the **`uvm_object`** type, and **`uvm_object`** is the base class type for all UVM components and transactions (not counting the **`uvm_void`** type).[1]

3.1. Class parameter types: uvm_sequence_item & int

The default transaction type for UVM components parameterized to a transaction type and the **`uvm_sequence`** type is the **`uvm_sequence_item`** type. Example component types that are parameterized to **`uvm_sequence_item`** include **`uvm_driver`** and **`uvm_sequencer`**. All user transactions will be derivatives of the **`uvm_sequence_item`** type.

A complete list of the eight UVM classes that are parameterized to the **`uvm_sequence_item`** type is shown in Appendix A on page 59.

The default type for many of the other UVM base class types parameterized to a transaction type is the 32-bit, 2-state **`int`** type.

NOBODY would ever us the **`int`** type as a transaction type. The **`int`** type is just the default, type-based, place holder inside of parameterized classes to make sure the class-based UVM library will compile correctly. EVERYBODY replaces the **`int`** type, typically with a class-based transaction type. Examples of commonly used components that are parameterized to the **`int`** type include **`uvm_tlm_fifo`** and **`uvm_analysis_tlm_fifo`**.

A complete list of the of the 69 UVM base classes that are parameterized to the **`int`** type is also shown in Appendix A on page 59.

3.2. UVM transactions

When approaching class-based verification for the first time a verification engineer is tempted to create one transaction type for the inputs and another transaction type for the outputs, because verification engineers who have done directed testing are accustomed to sending inputs into the design and then sampling the outputs for verification purposes.

When comparing UVM transactions to directed testing methods, transactions have fields for both inputs and outputs in the same transaction, while directed testing separates the input fields from the output fields. This is an important point when initially learning class-based verification.

[1] **`uvm_void`** is the root base class for all UVM components and transactions, but it is an empty virtual class that is extended to create the **`uvm_object`** base class. Nobody works with **`uvm_void`** but **`uvm_object`** is extensively used within all UVM testbenches.

In a UVM testbench environment the agent includes both a driver and a monitor. Even though the driver is given a copy of an entire transaction object that includes both inputs and outputs, the driver collects and only sends the transaction inputs to the design under test (DUT). The transaction outputs are ignored by the driver. The driver is one of the testbench components that must extract and properly send the individual input signals to the DUT.

The monitor actually samples both inputs and outputs from the DUT interface. The driver side agent has a monitor that will sample both the inputs and outputs but only the inputs will be processed by to the predictor inside of a scoreboard as noted in Figure 1. The sampled outputs are still in that transaction but they are completely ignored.

The output-side agent uses the exact same monitor, which samples both the inputs and the outputs from the DUT interface, but on the output-side monitor, even though both the inputs and outputs have been sampled and sent to the scoreboard, the inputs will be discarded by the comparator in the scoreboard as noted in Figure 1 and the actual DUT outputs will be used for comparison against the predicted outputs.

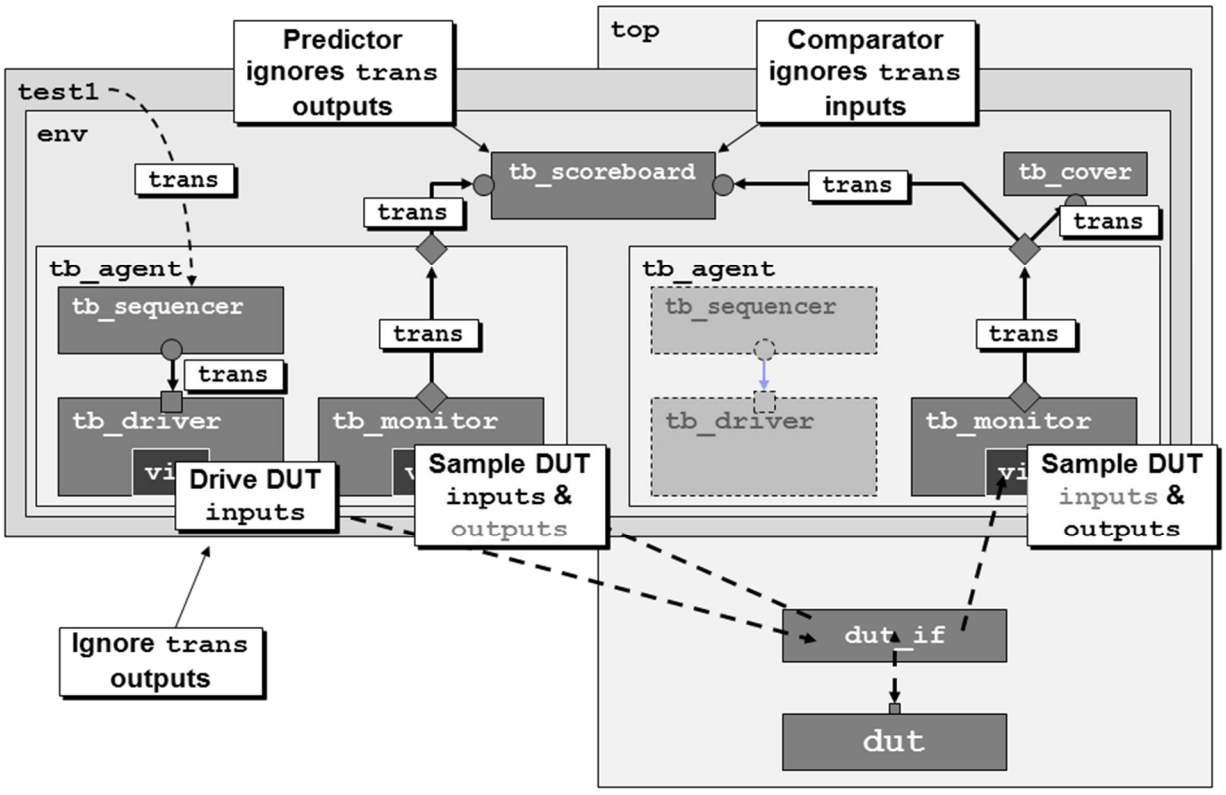

Figure 1 - Transaction passing

The novice UVM verification user is tempted to create two different types of transactions, one that only holds sampled inputs and another that only holds sampled outputs, but if two different transaction types are used in two different monitors it means that the agent is not reusable on

both the stimulus-driving and output-sampling sides of the environment. By sampling both inputs and outputs in the same monitor and discarding the outputs on the stimulus-driving side and the inputs on the output-sampling side, we can reuse the exact same monitoring agents.

Understanding this technique explains how the same transaction can be used on both the input-side and output-side of the verification environment. This technique is typically unknown to engineers who have only done Verilog directed testbenches in the past.

3.3. Unnecessary Output Randomization

If the transaction class has separately defined output and input variables, there is no need to randomize the output variables.

The randomized inputs will be sent to the DUT while any randomized outputs would just be discarded; hence, randomization of outputs would just be an inefficient additional simulation step.

4. UVM transaction types

As previously mentioned, all user-defined transaction types should be extended from the **`uvm_sequence_item`** type, and the **`uvm_sequence_item`** class type is a derivative of the **`uvm_object`** class type.

4.1. Standard class formatting

Although not required by UVM, I prefer to follow a standard code-layout for my UVM testbench components and UVM transaction definitions. A standard format helps with the readability of the code and helps me to quickly find important sections of the code. The formatting steps and order that I follow are shown in Figure 2.

```
(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  | (if any)
(3)  Declare virtual interface        | (if any)
(4)  Declare ports & components       | (if any)
(5)  Standard new() constructor       |
(6)  build_phase()                    | (if any)
(7)  connect_phase()                  | (if any)
(8)  Other pre-run phases             | (if any)
(9)  run_phase()                      | (if any)
(10) Other post-run phases            | (if any)
(11) Common component methods         | (if any)
```

Figure 2 - Standard class formatting

As noted, the above format and order is not only used for transactions but also for testbench components. For transactions, there are no phase methods, so my preferred order looks like this:

Using Field Macros

```
(0)  Declare transaction variables
(1)  Register class with factory
     and declare field macros
(2)  Declare vars & covergroups
(5)  Standard new() constructor
(11) Common transaction methods
     convert2string() method
```

Figure 3 - Transaction formatting w/ field macros

Using do_methods()

```
(1)  Register class with factory

(2)  Declare vars & covergroups
(5)  Standard new() constructor
(11) Common transaction methods
     convert2string() methods
     do_copy() / do_compare() /
     other do_methods()
```

Figure 4 - Transaction formatting w/ do_methods()

The differences between using field macros and **`do_methods()`** will be described in later sections.

5. Transaction class methods

One of the advantages of using transaction classes is that they can contain important utility methods. These important methods remove many coding requirements that existed in Verilog testbenches.

There are two ways to implement important transaction methods: the first is to use field macros, the second is to use manual coding techniques by overriding the built-in **`do_methods()`**.

Using field macros is relatively simple but they can be inefficient during simulation and difficult to debug if something does go wrong. The UVM User Guide[8] was largely written by Cadence UVM experts and Cadence recommends using these field macros. Mentor UVM experts typically recommend that verification engineers avoid using the field macros due to their coding and simulation inefficiencies.[1][3][5][12]

Unfortunately, the UVM User Guide only documents the use of field macros and does not include any documentation about an alternate approach, that of using the **`do_methods()`** to define the standard transaction methods. Similarly, the Verification Academy [12] only shows the use of **`do_method()`** overrides to define the standard transaction methods and does not demonstrate the alternate approach of using field macros. Verification engineers that reference these two sources are often perplexed about the divergent recommendations and this becomes a source of much confusion to novice UVM users. It would have been better if the two major sources of information had promoted their preferred approach and then acknowledge that there was an alternate method. Adam Erickson's paper on "Evil Macros"[1] discusses both approaches and promotes the use of the **`do_methods()`**, while many of my professional colleagues prefer the ease-of-use of the field macros.

Overriding the built-in **`do_methods()`** requires more manual coding by the verification engineer but the overridden **`do_methods()`** are more simulation efficient and not too difficult to code once a few important techniques are understood.

Both of these techniques will be described in later sections.

6. Introduction to standard transaction methods

In this paper, the important convenience functions are referred to as standard transaction methods.

The standard transaction methods are zero-time functions that should be defined in a transaction class and should always include user defined **`copy()`**, **`compare()`** and **`convert2string()`** methods. One other method that is important to define is the **`print()`** function, just because many users expect it to be available, even though **`convert2string()`** is often both more simulation and more print-space efficient.

If the design includes serial-to-parallel or parallel-to-serial activities that are very common among network packet-based designs, additional functions that will be included in the standard transaction function list, include: **`pack()`** and **`unpack()`**. One other standard transaction function is the **`record()`** function that is somewhat tool specific and used to help debug transient transaction objects.

The user should never override the standard transaction methods directly, but instead should indirectly define the required methods by overriding the base class **`do_methods()`** or by implementing field macros.

Each user transaction class that extends from **`uvm_sequence_item`** inherits the standard transaction methods, which are mostly-empty methods defined in the **`uvm_object`** virtual base class. One or more of these methods should be either directly or indirectly defined in the user transaction class.

6.1. Factory registration of transactions

The user's transaction class must be registered with the factory.

If you are going to create the standard transaction methods by overriding the built-in **`do_methods()`** you must use the **`` `uvm_object_utils() ``** macro.

If you are going to create the standard transaction methods by using field macros, you must use the **`` `uvm_object_utils_begin() ``** / **`` `uvm_object_utils_end ``** macros.

What is different about these macros? The details are described in `uvm_object_utils() -vs- `uvm_object_utils_begin()/_end section.

6.2. `uvm_object_utils() -vs- `uvm_object_utils_begin()/_end

In the UVM **`src/macros/uvm_object_define.svh`** file, there exists two forms of **`` `uvm_object_utils() ``** macros to register the transaction with the factory, along with other important transaction class based setup. The pertinent code is shown below.

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

Figure 5 - Actual `define uvm_object_utils macro definition

The first observation to make is that calling **`uvm_object_utils(T)** is equivalent to calling the back-to-back commands **`uvm_object_utils_begin(T)** / **`uvm_object_utils_end**. If a transaction is defined using the **`uvm_object_utils(T)** macro, no field macros are permitted in the transaction class definition. This is the technique recommended by Mentor UVM experts.[1][3]

As shown below, the **`uvm_object_utils_begin(T)** macro actually implements some important user-transaction functionality, including:

`m_uvm_object_registry_internal	- register the transaction class with the factory
`m_uvm_object_create_func	- define the `create()` method for this class
`m_uvm_get_type_name_func	- define the `get_type_name()` method for this class
`uvm_field_utils_begin()	- prepares to process defined field macros, if used

The actual **`uvm_object_utils_begin(T)** macro definition is shown in Figure 6.

```
`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)
```

Figure 6 - Actual `define uvm_object_utils_begin macro definition

If the **`uvm_object_utils()** macro is used, the **`uvm_field_utils_begin()** macro, which prepares the appropriate setup code for using field macros, is not populated with any field macros. As stated earlier, the **`uvm_object_utils()** macro should only be used if important transaction class methods are defined by overriding the **do_*methods()***.

The **`uvm_field_utils_begin()** macro defines a few functions important to field macros then opens a function definition that will be populated by field macros, if used.

The **`uvm_object_utils_end** macro simply closes off the **`uvm_field_utils_begin()** macro using a macro name that intuitively finishes the **field_utils** block. The actual (and trivial) **`uvm_object_utils_end** macro definition is shown in Figure 7.

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

Figure 7 - Actual `define uvm_object_utils_end macro definition

Unfortunately, because there are two different macros to register the transaction with the factory, there are also two different coding styles that are commonly used to define transactions and the style chosen depends on whether **`do_method()`** overrides or field macros are employed.

When I use the **`do_method()`** style, the transaction class definition resembles this:

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

When I use the field macros style, the transaction class definition resembles this:

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

It is annoying that I must use two different **`uvm_object_utils()`** placements just because I choose to use **`do_methods()`** or field macros, but the field macro style requires the variables to be declared before they are referenced by field macros, where the declared field macros must be encapsulated within the **`uvm_object_utils_begin(T)`** / **`uvm_object_utils_end`** pair.

It is certainly possible to place the **`uvm_object_utils()`** macro call after declaring variables when using the **`do_methods()`** style, but I prefer to see my **`uvm_object_utils()`** command at the top of the class definition, just beneath the class header, just as I do for all testbench component and sequences classes.

What I really want is a pair of macros to encapsulate the field macros without requiring that they be placed within a **`uvm_object_utils_begin(T)`** / **`uvm_object_utils_end`** pair, perhaps macros called **`uvm_field_utils_begin()`** / **`uvm_field_utils_end`**. From Figure 6 shown on page 13, I saw that these macros already existed! So I tried placing the **`uvm_object_utils()`** macro at the top of the transaction class, declared variables, and tried using **`uvm_field_utils_begin()`** with field macro declarations, as shown in Figure 8.

```
class trans1 extends uvm_sequence_item;
  `uvm_object_utils(trans1)
  rand bit [7:0] q;
  rand bit [7:0] a, b, c;

  `uvm_field_utils_begin(trans1) // ** Error this line
    `uvm_field_int(q, UVM_ALL_ON)
    `uvm_field_int(a, UVM_ALL_ON)
    `uvm_field_int(b, UVM_ALL_ON)
    `uvm_field_int(c, UVM_ALL_ON)
  `uvm_field_utils_end
  ...
```

Figure 8 - Illegal Syntax - Calling both `uvm_object_utils() and `uvm_field_utils_begin()

Unfortunately, this did not work and the compiler reported the error:

```
** Error: '__m_uvm_field_automation' already exists;
          must not be redefined as a function.
```

The problem is that the `` `uvm_object_utils() `` macro also calls the `` `uvm_field_utils_begin(T) `` macro, and since the `` `uvm_field_utils_begin(T) `` macro defines the `__m_uvm_field_automation` function, the function is defined twice, which is illegal.

6.3. __m_uvm_field_automation() method

The `` `uvm_object_utils_begin() `` macro, defined in the `uvm/src/macros/uvm_object_defines.svh` file, defines the first 20 lines of an internal `__m_uvm_field_automation()` method and the `` `uvm_object_utils_end `` macro defines the last 7 lines of the same macro. If field macros are used to define the standard transaction methods, each field macro contributes to the middle section of the `__m_uvm_field_automation()` method. For example, each call to the `` `uvm_field_int() `` macro adds 59 more lines of code to the middle of the `__m_uvm_field_automation()` macro.

The 59-line block of code added to the `__m_uvm_field_automation()` method is mostly a very large `case()` statement that executes the proper code for the case values of: `UVM_CHECK_FIELDS`, `UVM_COPY`, `UVM_COMPARE`, `UVM_PACK`, `UVM_UNPACK`, `UVM_RECORD`, `UVM_PRINT` and `UVM_SETINT`. When the user calls the `compare()` method, the compare method actually calls the internally constructed `__m_uvm_field_automation()` method with the `UVM_COMPARE` argument to execute the `UVM_COMPARE` code in each of the added `case()` statements.

For each field macro defined, another large block of code is added to the middle of the internal`__m_uvm_field_automation()` method, and each block of code includes multiple calls to other methods within a `__m_uvm_status_container` class, so if there are ever any problems related to the field macros, the debugging task is extremely verbose and complex. Fortunately, the field macros work properly most of the time, but when they don't work, debugging is time-consuming and extremely frustrating.

6.4. Proposed Future UVM Macro Change

It seems that the previous `` `uvm_field_utils_begin(T) `` macro problem described in section 6.2 could be easily fixed by modifying the definition for the `` `uvm_object_utils(T) `` macro. Instead of calling `` `uvm_object_utils_begin(T) `` / `` `uvm_object_utils_end ``, which calls four other macros, redefine `` `uvm_object_utils() `` to just call three of the macros, omitting the call to the `` `uvm_field_utils_begin() `` macro, which appears to be completely unnecessary in a non-field macros transaction class definition. The newly proposed definition is shown in Figure 9.

```
`define uvm_object_utils(T)                \
   `m_uvm_object_registry_internal(T,T) \
   `m_uvm_object_create_func(T)          \
   `m_uvm_get_type_name_func(T)
```

Figure 9 - Proposed UVM Change - new definition for `uvm_object_utils(T)

If it is determined that there are no backward compatibility issues, I request that the UVM Standards Committee implement this change. *Time to step off the soap box and get back to technical usage detail.*

7. Inherited standard transaction methods

The user's transaction class is extended from the **`uvm_sequence_item`** class, which is derived from the top-level **`uvm_object`** class type. Through this inheritance path, the user's transaction class inherits the following important utility methods:

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

Figure 10 - Important, inherited utility non-virtual methods

These 11 standard transaction methods are non-virtual functions or non-virtual void functions and the user should NEVER extend or override any of these important utility methods in a transaction class. These standard transaction methods execute a large amount of UVM overhead code and then call the **`__m_uvm_field_automation()`** method (which executes operations built from user-declared field macros) followed by calling **`do_methods()`**, which can be overridden by the user, as shown in Figure 11.

As described in the preceding paragraph, it is important to note that calling any of the standard transaction methods actually executes both field macro code *AND* the corresponding **`do_methods()`**. The significance of this fact is that an engineer can properly define field macros and then exclude the implementation of field macro functionality if that functionality is subsequently implemented using the corresponding **`do_methods()`**. Conversely, it is very risky to implement any of the standard transaction methods by combining partial implementation using field macros and completing the implementation with a partial-functionality definition in a **`do_method()`**. The latter is never seen in standard industry practice and is highly discouraged.

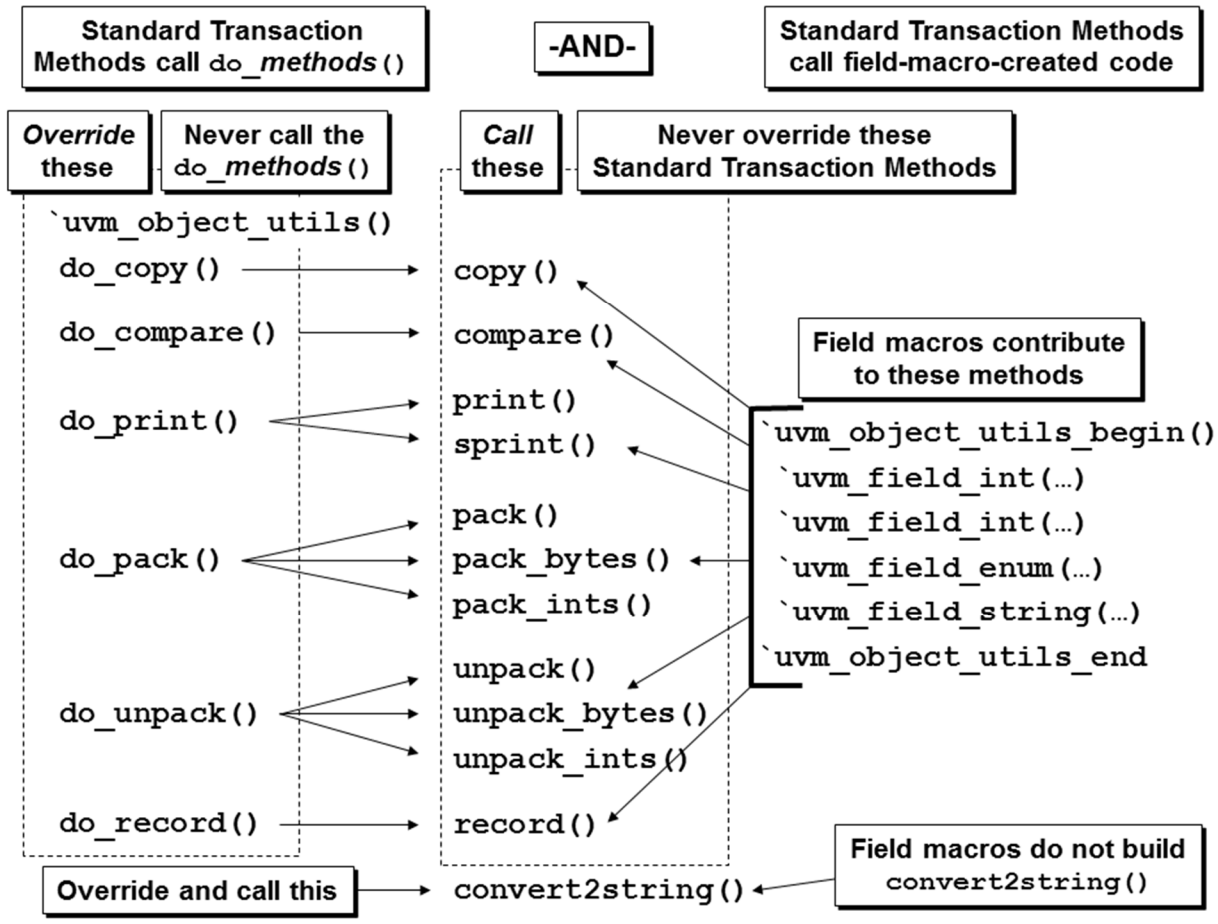

Figure 11 - Standard transaction methods - two ways to create them

The actual prototypes for these 11 methods are shown in Figure 12.

```
function void   copy       (uvm_object   rhs);
function bit    compare    (uvm_object   rhs, uvm_comparer comparer=null);
function void   record     (                  uvm_recorder recorder=null);
function void   print      (                  uvm_printer  printer =null);
function string sprint     (                  uvm_printer  printer =null);
function int pack          (ref   bit           bitstream [],
                            input uvm_packer    packer=null);
function int pack_bytes    (ref   byte unsigned bytestream[],
                            input uvm_packer    packer=null);
function int pack_ints     (ref   int  unsigned intstream [],
                            input uvm_packer    packer=null);
function int unpack        (ref   bit           bitstream [],
                            input uvm_packer    packer=null);
function int unpack_bytes  (ref   byte unsigned bytestream[],
                            input uvm_packer    packer=null);
function int unpack_ints   (ref   int  unsigned intstream [],
                            input uvm_packer    packer=null);
```

Figure 12 - Important utility non-virtual method prototypes

7.1. Should I override the standard transaction methods?

So why do I have to declare field macros or override **`do_methods()`** to help implement the non-virtual methods shown in Figure 10. Why don't I just override these standard transaction methods directly and bypass the field macros and **`do_methods()`**?

Are you kidding? Have you seen the code for these methods in the **`uvm_object.svh`** *file??*

WARNING: you should not take any time to try to read, examine or figure out the following 69 lines of **`compare()`** code from the **`uvm_object`** base class. It is inserted into this paper to *discourage you from ever* considering the option to override the built-in **`compare()`** method. You should either use field macros or override the **`do_compare()`** method.

```
 1 // compare
 2 // -------
 3
 4 function bit  uvm_object::compare (uvm_object rhs,
 5                                     uvm_comparer comparer=null);
 6   bit t, dc;
 7   static int style;
 8   bit done;
 9   done = 0;
10   if(comparer != null)
11     __m_uvm_status_container.comparer = comparer;
12   else
13     __m_uvm_status_container.comparer = uvm_default_comparer;
14   comparer = __m_uvm_status_container.comparer;
15
16   if(!__m_uvm_status_container.scope.depth()) begin
17     comparer.compare_map.clear();
18     comparer.result = 0;
19     comparer.miscompares = "";
20     comparer.scope = __m_uvm_status_container.scope;
21     if(get_name() == "")
22       __m_uvm_status_container.scope.down("<object>");
23     else
24       __m_uvm_status_container.scope.down(this.get_name());
25   end
26   if(!done && (rhs == null)) begin
27     if(__m_uvm_status_container.scope.depth()) begin
28       comparer.print_msg_object(this, rhs);
29     end
30     else begin
31       comparer.print_msg_object(this, rhs);
32       uvm_report_info("MISCMP",
33            $sformatf("%0d Miscompare(s) for object %s@%0d vs. null",
34              comparer.result,
35              __m_uvm_status_container.scope.get(),
36              this.get_inst_id()),
37              __m_uvm_status_container.comparer.verbosity);
38       done = 1;
39     end
40   end
41
```

```
42   if(!done && (comparer.compare_map.get(rhs) != null)) begin
43     if(comparer.compare_map.get(rhs) != this) begin
44       comparer.print_msg_object(this, comparer.compare_map.get(rhs));
45     end
46     done = 1;  //don't do any more work after this case, but do cleanup
47   end
48
49   if(!done && comparer.check_type && (rhs != null) &&
50     (get_type_name() != rhs.get_type_name())) begin
51     __m_uvm_status_container.stringv = { "lhs type = \"", get_type_name(),
52                       "\" : rhs type = \"", rhs.get_type_name(), "\""};
53     comparer.print_msg(__m_uvm_status_container.stringv);
54   end
55
56   if(!done) begin
57     comparer.compare_map.set(rhs, this);
58     __m_uvm_field_automation(rhs, UVM_COMPARE, "");// LINE 58-field macros
59     dc = do_compare(rhs, comparer);               // LINE 59-do_compare()
60   end
61
62   if(__m_uvm_status_container.scope.depth()==1)  begin
63     __m_uvm_status_container.scope.up();
64   end
65
66   if(rhs != null)
67     comparer.print_rollup(this, rhs);
68   return (comparer.result == 0 && dc == 1);
69 endfunction
```

Figure 13 - UVM 1.1d - src/base/uvm_object.svh - compare() method implementation

REMINDER: you should not take any time to try to read, examine or figure out the preceding 69 lines of `compare()` code. It is inserted into this paper to ***discourage you from ever*** considering the option to override the built-in `compare()` method. You should either use field macros or override the `do_compare()` method. Anybody who tries to correctly override the built-in `compare()` method either needs to get-a-life or get-a-hobby! *(Writing this paper makes me think that I need to get-a-life!!)*

From the code in Figure 13, it can be seen that the default `compare()` method will make a call to implement the field macros (red-highlighted code on line 58) and will also call the user-defined `do_compare()` method (red-highlighted code on line 59). Your job is to either define field macros or override the `do_compare()` method and they will be automatically called by callbacks from the `compare()` method.

The problem you face if you try to override the `compare()` code is that there are 57 lines of important code before you either call the field macros on line 58, or call your implementation of the `do_compare()` method on line 59 (both of which are embedded in an internal `if`-statement). Then you still need to add 10 more lines of code after field macros or `do_compare()`. This means you cannot simply make a call to `super.compare()`. You would need something like a call to `super.pre_59_lines_compare()`, add you compare code, then call something like a `super.post_10_lines_compare()`, which of course is ridiculous!

There is similarly cryptic **uvm_object** base class code for the other important standard transaction methods. The value of the existing field macros and **do_*method()*** callbacks should now begin to be more obvious!

Guideline: do not directly override the **copy()**, **compare()** and other **uvm_object** base class standard transaction methods.

7.2. Inherited transaction utility methods

The user's transaction class, ultimately derived from the top-level **uvm_object** class type, also inherits the 3 important utility methods shown in Figure 14.

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

Figure 14 - Important, inherited utility virtual methods

These 3 methods are virtual functions or virtual void functions. The actual prototypes for these 3 methods are shown in Figure 15.

```
virtual function uvm_object create (string name="");
    return null;
endfunction

virtual function uvm_object clone ();
  uvm_object tmp;
  tmp = this.create(get_name());
  if (tmp == null) `uvm_warning("CRFLD", "... create failed ...")
  else             tmp.copy(this);
  return(tmp);
endfunction

virtual function string convert2string();
    return "";
endfunction
```

Figure 15 - Important utility virtual method prototypes

The virtual **do_*methods()*** will be described in later sections, but the **create()**, **clone()** and **convert2string()** methods are described in the next three sections.

7.3. create() method

Per the UVM Class Reference manual, "Every class deriving from **uvm_object**, directly or indirectly, must implement the create method."[7] When the **`uvm_object_utils(T)** macro is called, one of the actions of that macro is to automatically implement the **create()** method (the utils macro calls the **`m_uvm_object_create_func(T)** macro). If we do not call the **`uvm_object_utils()** macro, among other things, we would need to implement the **create()** method manually. A manual implementation example of the **create()** method from the UVM Class Reference manual is shown in Figure 16.

```
class mytype extends uvm_object;
  ...
  virtual function uvm_object create(string name="");
    mytype t = new(name);
    return t;
  endfunction
```

Figure 16 - uvm_object create() method - manual definition

Guideline: never manually implement the **`create()`** method. Call the **`\`uvm_object_utils()`** macro to automatically implement the **`create()`** method.

7.4. clone() method

By default, the **`clone()`** method calls the **`create()`** method (constructs an object of the same type) and then calls the **`copy()`** method. It is a one-step command to create and copy an existing object to a new object handle.

Guideline: never override the **`clone()`** method. The existing default behavior is good.

7.5. convert2string()

The **`convert2string()`** method is one of the most important methods to define within a transaction. In the absence of a **`convert2string()`** method, each user has to decide how to print transaction values at different locations in the testbench.

The default **`convert2string()`** method defined in the **`uvm_object`** virtual base class is basically a placeholder and just returns an empty string. The relevant code snippets for the **`uvm_object`** base class **`convert2string()`** method are shown in Figure 17.

```
extern virtual function string convert2string();
…
function string uvm_object::convert2string();
  return "";
endfunction
```

Figure 17 - uvm_object source code for convert2string()

It is a common courtesy that the designer of every transaction class should override the **`convert2string()`** method with a well formatted string of the transaction variables. The **`convert2string()`** method is more efficient than calling the transaction **`print()`** method, which has to format the variables into table or tree-like formats.

Guideline: Every user-defined transaction method should include a **`convert2string()`** method.

By creating a **`convert2string()`** method, the transaction class developer is providing, to anybody who uses the transaction objects, the ability to print out the transaction object contents without the trouble to create their own display-type command.

The `convert2string()` method returns a formatted string of the transaction object's field contents. The `convert2string()` method should be called from a message macro, which also includes an "`id`" string field and a verbosity setting.

7.6. Plan for extended convert2string() methods

The `convert2string()` method of a transaction class extended from a base transaction class will either need to reformat all of the base class `convert2string()` transaction variables (discouraged) or call super.`convert2string()` to pick up the string information for the base transaction variables (preferred).

When printing, I prefer to group transaction inputs together followed by grouped transaction outputs. If you call `super.convert2string()`, you will probably have the extended input and output signals mixed with the base class input and output signals.

To avoid a mixed order of inputs and outputs, I recommend the creation of two more transaction functions called `output2string()` and `input2string()`.

In the following example (Figure 18), `trans2` extends `trans1` and both classes have `input2string()` and `output2string()` methods. The extended class makes `super.`string-`method()` calls, concatenating extended variables to base class variables in the respective return statements:
intputs: `return ({super.input2string(), " " ,s});`
outputs: `return ({super.output2string(), " " ,s});`
The `trans2` transaction class has a very simple definition for `convert2string()`, which includes: `return ({output2string(), " ", input2string()});`
This way the inputs are grouped together and outputs are grouped together when printed.

```
class trans1 extends uvm_sequence_item;
  `uvm_object_utils(trans1)
       bit [7:0] a; // base output
  rand bit [7:0] b; // base input
  …
  function string input2string();
    return($sformatf("b=%2h", b));
  endfunction

  function string output2string();
    return($sformatf("a=%2h", a));
  endfunction

  function string convert2string();
    return ({input2string(), "  ", output2string()});
  endfunction
endclass

class trans2 extends trans1;
  `uvm_object_utils(trans2)
       bit [7:0] c; // extended output
```

```
  rand bit [7:0] d; // extended input
  …
  function string input2string();
    string s;
    s = $sformatf("d=%2h", d);
    return ({super.input2string()," ",s});
  endfunction

  function string output2string();
    string s;
    s = $sformatf("c=%2h", c);
    return ({super.output2string()," ",s});
  endfunction

  function string convert2string();
    return ({input2string(), " ", output2string()});
  endfunction
endclass
```

Figure 18 - Extended transaction function calls to super.output2string() & super.input2string()

7.7. Transaction printAll() method??

There are some examples in industry where the creators of transaction classes also create built-in **`printAll()`** methods that can be called directly without the need to call the **`convert2string()`** method from a **`` `uvm_info() ``** macro. This is not recommended because although inserting a **`printAll()`** method into the transaction would certainly make printing transaction information easier, it unfortunately also semi-permanently fixes the "**`id`**" string and verbosity setting.

The **`convert2string()`** method returns a string value that should be called from a **`` `uvm_info() ``**, **`` `uvm_error() ``** or **`` `uvm_fatal() ``** message macro.[2]

There are times when you will want to report debug information and you will want to print transaction values with a verbosity setting of **`UVM_DEBUG`**. At other times you will want the transaction values to print using the default **`UVM_MEDIUM`** verbosity setting, while at other times you will want to only print successful transaction values if you enable the **`UVM_HIGH`** verbosity setting. It is also useful to use unique "**`id`**" values in different places so that printing of some transactions can be masked while printing of other transactions can be promoted to always print.

If field macros are used, the built-in **`print()`** method will be populated, but when using the **`print()`** method the printed values will again be largely unmaskable and printed in a somewhat verbose multi-line table or tree format. For these reasons and for better verbosity control, I tend to skip the **`print()`** method in favor of the **`convert2string()`** method.

[2] The **`` `uvm_warn ``** message macro is almost worthless because it has a verbosity setting of **`UVM_NONE`** and is therefore difficult to suppress. I prefer to use **`` `uvm_info ``** message macros with different verbosity settings to replace the cumbersome **`` `uvm_warn ``** message macro.

8. Do_methods()

As was mentioned earlier, there are two ways to implement the important standard transaction methods. The standard transaction methods can be implemented using either the field macros or by overriding the built-in `do_methods()` shown in Figure 20. This section describes the implementation of the standard transaction methods by overriding the `do_methods()`.

The 11 standard transaction methods shown in Figure 19 can be implemented by overriding the 6 `do_methods()` also shown in Figure 19. The `do_methods()` are empty callback methods defined in the `uvm_object` base class. The user should never directly call any of the `do_methods()`. The `do_methods()` are called by the like-named, 11 standard transaction methods that are inherited from the `uvm_object` base class.

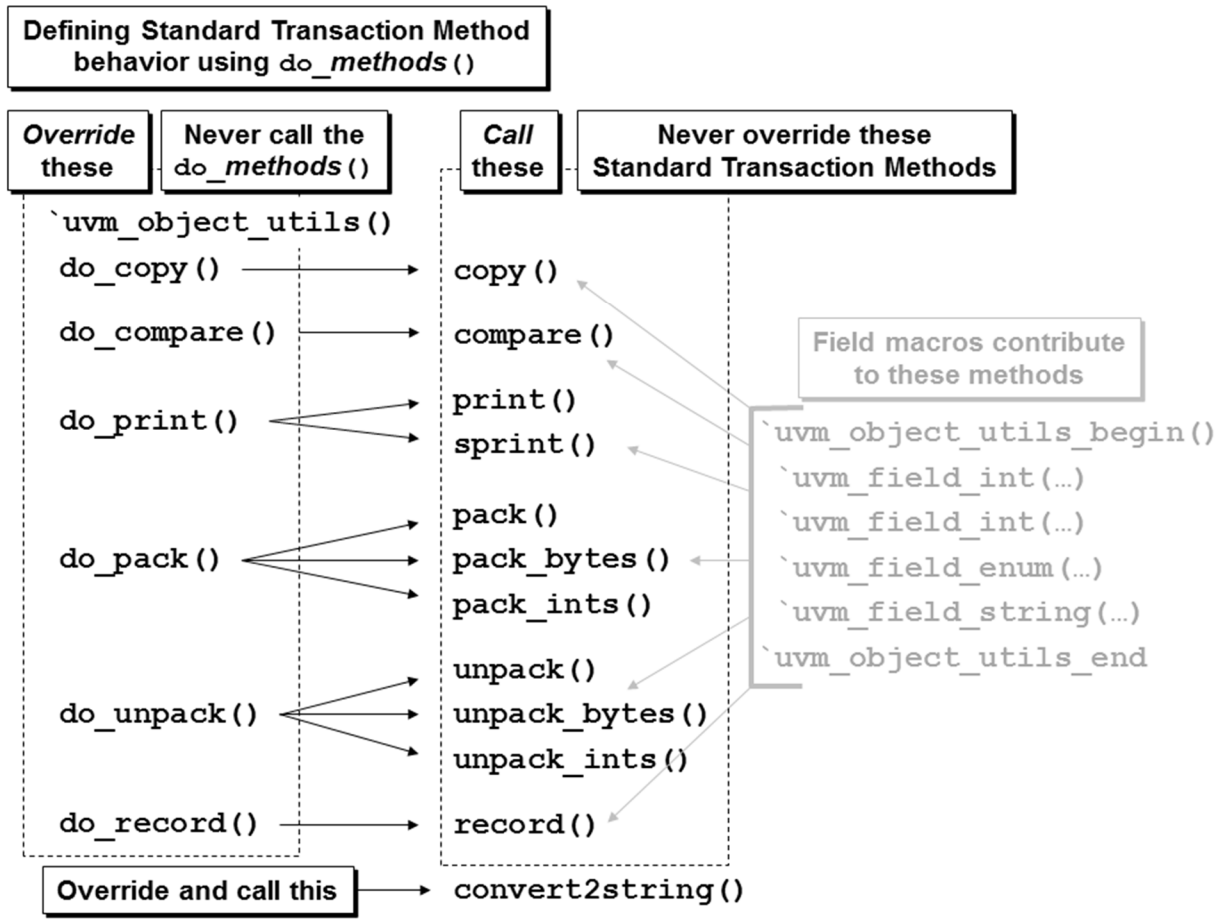

Figure 19 - Creating the standard transaction methods by overriding the built-in do_methods()

The user can override the built-in `do_methods()`, shown in Figure 20, which will affect how the standard transaction methods behave when called.

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

Figure 20 - Inherited do_*method()* hooks to define standard transaction methods

8.1. Virtual method rules and virtual do_*method()* prototypes

All of the user-overridable `do_methods()` are virtual methods, and SystemVerilog virtual methods have strict argument compatibility rules. When extending a SystemVerilog class and overriding a virtual method in an extended class, all argument types, names and return types must match the base class virtual method argument types and names, which means the method argument types and names cannot be changed.

This is simply a rule of object oriented languages like SystemVerilog and has nothing to do with the UVM methodology. UVM users must simply follow SystemVerilog rules and this is one of those rules.

Since all of the `do_methods()` in the `uvm_object` base class are virtual methods, overriding those methods in the user transaction class requires the user to use the exact same argument types and names.

8.2. base-class casting to extended class handle

Nonspecific to UVM is the concept of assigning a base class handle to an extended class handle. Although it is a somewhat side-topic, it is an important topic when using UVM so it discussed in this section.

SystemVerilog permits direct assignment of an extended handle to a base handle. There might be multiple different extensions of the same base class type, and each extension can add unique variables and define different unique methods in the extended class. Since any of these extended class handles can be assigned to the base class handle, the newly assigned base handle cannot call the extended methods and variables that were added to extended classes since those variables and methods could be different from assignment to assignment and the base class can only guarantee existence of base methods and variables.

On the other hand, SystemVerilog does NOT permit direct assignment of a base class handle to a derived class handle because the derived class typically expects to access more variables and methods than existed in the base class definition and if the base class handle was assigned from a completely different extended object, the expected methods and variables might not exist. The base class handle type has no knowledge of the extended variables and methods.

If a constructed extended class object is assigned to a base class handle, the handle type is converted to the base class handle type and access to extended methods and extended variables is

not possible using the base class handle, even though the methods and variables still exist. If this base class handle is then **$cast**[3] back to a declared extended class handle, then we again have access to the original variables, their values, and the extended methods. This is an important technique used with UVM standard **do_*methods()***.

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

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

Figure 21 - Overriding the do_copy() and do_compare() methods with uvm_object inputs

In Figure 21, the first three lines of the **do_copy()** method and three of the first four lines of the **do_compare()** method are standard required code. In all of the **do_*methods()***, the first argument of the prototype header is an input of the **uvm_object** base class handle type, but when each **do_*method()*** is called, they will be passed an extended **trans1** transaction class handle, which will convert the **trans1** transaction class handle into the **rhs uvm_object** base class handle type.

Once a **trans1** class handle has been converted into a **uvm_object** base-class handle type, it is necessary to (1) declare a handle of the **trans1** (derivative of **uvm_object**) handle type, and then (2) **$cast** the **uvm_object** base class handle-type back into the **trans1** (derivative) class handle type, to recover all of the transaction variables and gain access to the transaction methods that were hidden when the transaction handle was converted into a **uvm_object** handle.

This is why the first few lines of each UVM standard **do_*method()*** might look strange. This **$cast**ing is simply a required step to recover all of the variables and methods of a transaction type, and is just SystemVerilog overhead code required by the UVM standard **do_*methods()***.

8.3. rhs & rhs_ do_*method()* arguments

There are many industry example implementations of the **do_*methods()*** where the **trans1** (or equivalent) transaction class handle is declared with the handle name **rhs_** as shown in Figure 22. Then the **do_*method()*** input argument **rhs** is **$cast** to the **trans1 rhs_** handle. I

[3] $cast performs checking. If the base class is holding a handle to a derived type that is different than the type being assigned, $cast will fail. When $cast is called as a task, this failure results in a tool-generated error message. When $cast is called as a function, the failure results in a return status of 0 (a success returns 1).

personally believe this adds confusion to the code. It is too easy for the reader to miss the trailing "_" on the **rhs_** handle and make incorrect assignments and assumptions.

```
function void do_copy(uvm_object rhs);
  trans1 rhs_;
  $cast(rhs_, rhs);
  if(!$cast(rhs_, rhs)) `uvm_fatal("trans1", "ILLEGAL do_copy() cast")
  a = rhs_.a;
  … (copy remaining variables)
endfunction
```

Figure 22 - Common do_copy() coding example with trans1 declared using rhs_ handle name

The **uvm_object** handle name of **rhs** in each of the standard transaction methods prototypes cannot be modified, but the commonly used transaction **rhs_** handle name can be changed. I prefer to replace the **rhs_** handle name with **tr** as shown in Figure 23, which is visibly distinct from the input **rhs** handle name.

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

Figure 23 - Preferred do_copy() coding example with trans1 declared using tr handle name

I believe the code is more readable and less error prone by using the distinct **tr** handle name.

8.4. uvm_object default do_*methods*()

The UVM top-level base class (at least the one we care about) is the **uvm_object** class type. The **uvm_object** virtual base class includes the following empty void virtual functions:

do_copy() (Figure 26),
do_print() (Figure 29),
do_record() (Figure 30),
do_pack() (Figure 31),
do_unpack() (Figure 32).

The **uvm_object** virtual base class also includes one almost-empty status-returning virtual function:

do_compare() (Figure 33).

8.5. copy() and do_copy()

The built-in **copy()** method executes the **__m_uvm_field_automation()** method with the required copy code as defined by the field macros (if used) and then calls the built-in **do_copy()** virtual function. The built-in **do_copy()** virtual function, as defined in the **uvm_object** base

class, is also an empty method, so if field macros are used to define the fields of the transaction, the built-in `copy()` method will be populated with the proper code to copy the transaction fields from the field macro definitions and then it will execute the empty `do_copy()` method, which will perform no additional activity.

The `copy()` method can be used as needed in the UVM testbench. One common place where the `copy()` method is used is to copy the sampled transaction and pass it into a `sb_calc_exp()` (scoreboard calculate expected) external function that is frequently used by the scoreboard predictor[2] as shown in Figure 24.

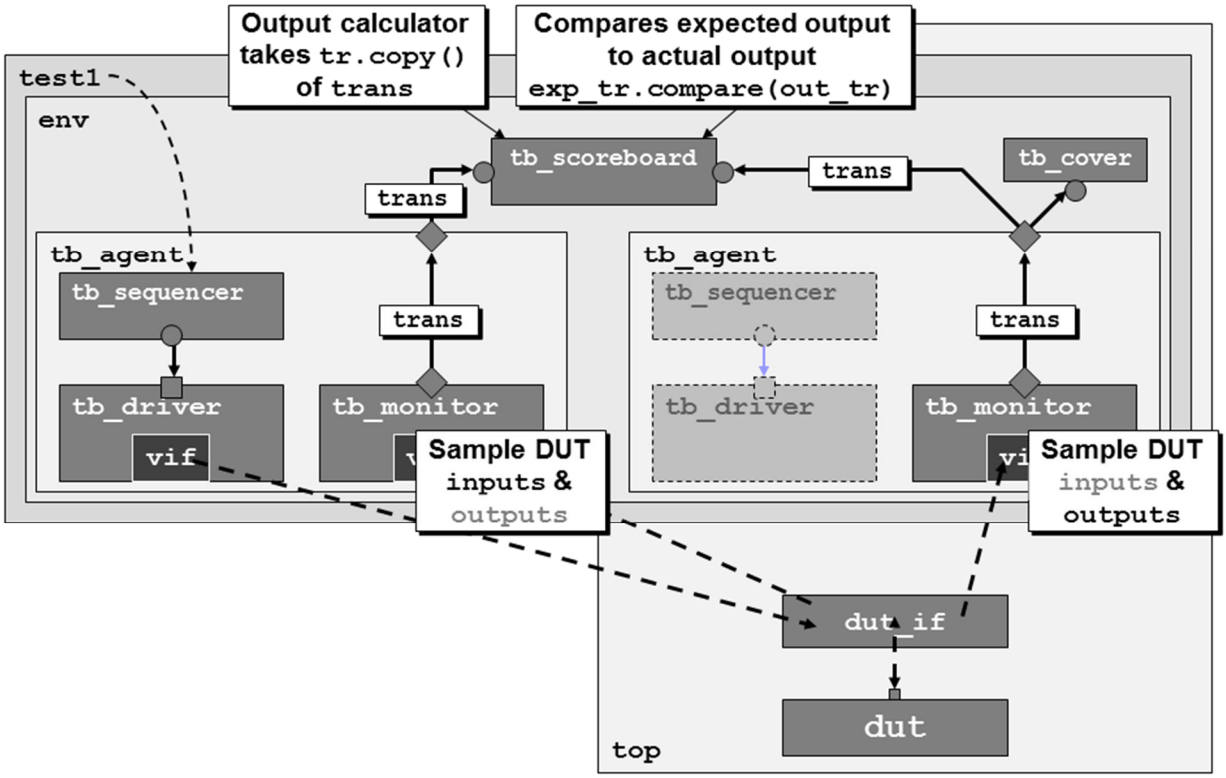

Figure 24 - Transaction copy() and compare() methods - common usage block diagram

An example usage of the `copy()` method is shown in the scoreboard calculate-expected function of the `sb_predictor::sb_calc_exp()` function in Figure 25. The transaction is passed through a `uvm_analysis_port` (originating from the `tb_monitor` in the `tb_agent`) to the `sb_calc_exp()` method in the `sb_predictor` located inside the `tb_scoreboard` class. The `t` transaction is then copied to a locally declared and created (line 4) transaction object (`tr.copy(t);` on line 14), then the calculated output value `dout` is copied to the transaction `dout` variable (`tr.dout = dout;` on line 18) and returned to the calling `sb_predictor` component (line 19).

```
 1 function trans1 sb_predictor::sb_calc_exp (trans1 t);
 2   static logic [15:0] next_dout;
 3          logic [15:0] dout;
 4   trans1 tr = trans1::type_id::create("tr");
 5   //--------------------------
 6   `uvm_info(get_type_name(), t.convert2string(), UVM_HIGH)
 7   // async reset: reset the next_dout AND current dout values -OR-
 8   // non-reset  : assign dout values & calculate the next_dout values
 9   dout = next_dout;
10   if      (!t.rst_n) {next_dout,dout} = '0;
11   else if ( t.ld)     next_dout       = t.din;
12   else if ( t.inc)    next_dout++;
13   // copy all sampled inputs & outputs
14   tr.copy(t);
15   // overwrite the dout values with the calculated values.
16   // dout values were either calculated in the previous cycle
17   //      or asynchronously reset in this cycle
18   tr.dout = dout;
19   return(tr);
20 endfunction
```

Figure 25 - Example sb_predictor.sv - collecting transactions using the tr.copy() method

8.6. Using the copy() method: to_tr.copy(from_tr)

The **`copy()`** method copies values from the **`from_tr`** object to the variables in the **`to_tr`** object (you are copying the values of variables from another transaction into this transaction). The transaction handle that is used to call the method name holds the destination variables. The transaction handle that is passed as an argument to the method holds the source variable values.

The default **`do_copy()`** method defined in the **`uvm_object`** virtual base class is empty. The relevant code snippets are shown in Figure 26.

```
extern virtual function void do_copy(uvm_object rhs);
…
function void uvm_object::do_copy(uvm_object rhs);
  return;
endfunction
```

Figure 26 - do_copy() inherited virtual method prototype and source code

The **`trans1`** code with **`do_copy()`** method used with the **`sb_predictor`** class code of Figure 25 is shown in Figure 27.

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

       logic [15:0] dout;    // outputs not randomized
  rand bit   [15:0] din;
  rand bit          rst_n;
```

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

  function void do_copy(uvm_object rhs);
    trans1 tr;
    if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_copy() cast")
    dout  = tr.dout;
    din   = tr.din;
    rst_n = tr.rst_n;
  endfunction

  function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    trans1 tr;
    bit    eq;
    if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_compare() cast")
    eq  = super.do_compare(rhs, comparer);
    eq &= (dout === tr.dout);
    return(eq);
  endfunction

  function string input2string();
    return($sformatf("din=%4h  rst_n=%b", din, rst_n));
  endfunction

  function string output2string();
    return($sformatf("dout=%4h", dout));
  endfunction

  function string convert2string();
    return($sformatf({input2string(), "  ", output2string()}));
  endfunction
endclass
```

Figure 27 - trans1 example with do_copy() and do_compare() methods defined

8.7. print(), sprint() and do_print()

The built-in **`print()`** method is a void function that prints all of the field-macro defined fields in a table format by default. A **`print()`** method would only print the table header and footer if field macros are omitted and do_**`print()`** is not overridden by the user. Printing with the **`print()`** method is not tracked in the final UVM Report Summary because it cannot be called from the message macros with "**`id`**" string fields. Because the **`print()`** method is not called from the message macros, it also cannot be suppressed by using different UVM verbosity settings.

By contrast, the built-in **`sprint()`** method is a function that returns a multi-line formatted string with all of the defined fields in a table format (by default) and should be called from the message macros. Printing with the **`sprint()`** method ***is*** tracked in the final UVM Report Summary and since it is called from the message macros, it can be suppressed by using different UVM verbosity settings.

Since it is very likely that some user will attempt to call the transaction `print()` method, the next two either-or guidelines are recommended to avoid unexpected results.

Guideline: Implement the `print()` method using field macros.
-OR-
Guideline: Implement a `do_print()` method that returns the following string, "print() and sprint() are not implemented for this transaction type" as shown in Figure 28.

```
function void do_print(uvm_printer printer);
  $display("\n\n\t\t*** print() and sprint() are not implemented ",
           "for this transaction type ***\n\n");
endfunction
```

Figure 28 - NULL do_print() method

More important guidelines regarding transaction printing are shown below.

Guideline: Avoid using the `print()` method. Its output is verbose and cannot be suppressed by using UVM verbosity settings.

Guideline: Avoid using the `sprint()` method. Its output is verbose.

Guideline: If you do use one of the built-in printing methods, choose `sprint()` over `print()` and call it from a UVM message macro. Runtime verbosity settings can mask verbose `sprint()` method printouts if desired.

Guideline: Define and use the `convert2string()` method discussed in earlier sections. `convert2string()` is more simulation efficient, more print-space efficient and can be easily suppressed by using different runtime UVM verbosity settings.

The built-in `print()` and `sprint()` methods either implement the required code as defined by the field macros or they call the built-in `do_print()` virtual function. The built-in `do_print()` virtual function, as defined in `uvm_object`, is an empty method, so if field macros are used to define the fields of the transaction class, the built-in `print()` and `sprint()` methods will be populated with the proper printing code from most field macros and then they will execute the empty `do_print()` method, which will perform no additional activity.

The default `do_print()` method defined in the `uvm_object` virtual base class is empty. The relevant code snippets are shown in Figure 29.

```
extern virtual function void do_print(uvm_printer printer);
…
function void uvm_object::do_print(uvm_printer printer);
  return;
endfunction
```

Figure 29 - do_print() inherited virtual method prototype and source code

8.8. record() and do_record()

The built-in **record()** method executes the __m_uvm_field_automation() method with the required record code as defined by the field macros (if used) and calls the built-in **do_record()** virtual function. The built-in **do_record()** virtual function, as defined in the **uvm_object** base class, is also an empty method, so if field macros are used to define the fields of the transaction, the built-in **record()** method will be populated with the proper code to record the transaction fields from the field macro definitions and then it will execute the empty **do_record()** method which will perform no additional activity.

The default **do_record()** method defined in the **uvm_object** virtual base class is empty. The relevant code snippets are shown in Figure 30.

```
extern virtual function void do_record (uvm_recorder recorder);
…
function void uvm_object::do_record(uvm_recorder recorder);
  return;
endfunction
```

Figure 30 - do_record() inherited virtual method prototype and source code

8.9. pack() and do_pack()

The built-in **pack()**, **pack_bytes()**, and **pack_ints()** methods execute the __m_uvm_field_automation() method with the required packing code as defined by the field macros (if used) and then
they call the built-in **do_pack()** virtual function. The built-in **do_pack()** virtual function, as defined in the **uvm_object** base class, is an empty method, so if field macros are used to define the fields of the transaction class, the built-in **pack()**, **pack_bytes()**, and **pack_ints()** methods will be populated with the proper packing code from most field macro definitions and then they will execute the empty **do_pack()** method which, will perform no additional activity.

The default **do_pack()** method defined in the **uvm_object** virtual base class is empty. The relevant code snippets are shown in Figure 31.

```
extern virtual function void do_pack (uvm_packer packer);
…
function void uvm_object::do_pack (uvm_packer packer );
  return;
endfunction
```

Figure 31 - do_pack() inherited virtual method prototype and source code

8.10. unpack() and do_unpack()

Similarly, the built-in **unpack()**, **unpack_bytes()**, and **unpack_ints()** methods execute the __m_uvm_field_automation() method with the required unpacking code as defined by the field

macros (if used) and then they call the built-in **do_unpack()** virtual function. The built-in **do_unpack()** virtual function, as defined in the **uvm_object** base class, is an empty method, so if field macros are used to define the fields of the transaction class, the built-in **unpack()**, **unpack_bytes()**, and **unpack_ints()** methods will be populated with the proper unpacking code from most field macro definitions and then they will execute the empty **do_unpack()** method, which will perform no additional activity.

The default **do_unpack()** method defined in the **uvm_object** virtual base class is empty. The relevant code snippets are shown in Figure 32.

```
extern virtual function void do_unpack (uvm_packer packer);
…
function void uvm_object::do_unpack (uvm_packer packer);
  return;
endfunction
```

Figure 32 - do_unpack() inherited virtual method prototype and source code

8.11. compare() and do_compare()

The built-in **compare()** method executes the __m_uvm_field_automation() method with the required comparison code as defined by the field macros (if used) and then calls the built-in **do_compare()** virtual function. The built-in **do_compare()** virtual function, as defined in the **uvm_object** base class, is an empty method that returns a "**1**" ("true") value, so if field macros are used to define the fields of the transaction, the built-in **compare()** method will be populated with the proper code to compare the transaction fields from the field macro definitions and then it will perform an **and** operation with the "**1**" value returned from the **do_compare()** method, which will perform no additional activity.

The default **do_compare()** method defined in the **uvm_object** virtual base class is almost empty, but the default return value is **1** ("true"). The relevant code snippets are shown in Figure 33.

```
extern virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer);
…
function bit uvm_object::do_compare(uvm_object rhs, uvm_comparer comparer);
  return 1;
endfunction
```

Figure 33 - do_compare() inherited virtual method prototype and source code

The **compare()** method can be used as needed in the UVM testbench. One common and very important place where the **compare()** method is used is to compare the outputs of the expected transaction to the outputs of the actual transaction as shown in Figure 24.

An example usage of the **compare()** method is shown in the **run_phase()** task of the **sb_comparator** class in Figure 34. A forever-loop in the **run_phase()** task continuously gets the expected transaction from the predictor (**expfifo.get(exp_tr)**), then gets the output

transaction (`outfifo.get(out_tr)`), and then compares the output values from each transaction to each other (`out_tr.compare(exp_tr)`). Since the `compare()` method was properly defined using the `do_compare()` method, and only compares outputs and not inputs for this design in the `trans1` transaction class of Figure 27, the comparison in the scoreboard comparator is a very simple operation.

```
class sb_comparator extends uvm_component;
  `uvm_component_utils(sb_comparator)

  int VECT_CNT, PASS_CNT, ERROR_CNT;

  uvm_analysis_export   #(trans1) axp_in;
  uvm_analysis_export   #(trans1) axp_out;
  uvm_tlm_analysis_fifo #(trans1) expfifo;
  uvm_tlm_analysis_fifo #(trans1) outfifo;

  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    axp_in  = new("axp_in",  this);
    axp_out = new("axp_out", this);
    expfifo = new("expfifo", this);
    outfifo = new("outfifo", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    axp_in.connect (expfifo.analysis_export);
    axp_out.connect(outfifo.analysis_export);
  endfunction

  task run_phase(uvm_phase phase);
    trans1 exp_tr, out_tr;

    forever begin
      expfifo.get(exp_tr);
      outfifo.get(out_tr);
      if (out_tr.compare(exp_tr)) PASS (exp_tr);
      else                        ERROR(exp_tr, out_tr);
    end
  endtask

  function void report_phase(uvm_phase phase);
    super.report_phase(phase);
    if (VECT_CNT && !ERROR_CNT)
      `uvm_info("COMPARATOR",
     $sformatf("\n\n\n*** TEST PASSED - %0d vectors ran, %0d vectors passed ***\n",
                VECT_CNT, PASS_CNT), UVM_LOW)
    else
      `uvm_error("COMPARATOR",
    $sformatf("\n\n\n*** TEST FAILED - %0d vectors ran, %0d vectors passed, %0d vectors failed ***\n",
                VECT_CNT, PASS_CNT, ERROR_CNT))
  endfunction
```

```
  function void PASS(trans1 exp_tr);
    `uvm_info("cmp vector",
      $sformatf("*** Vector Passed: %s ***", exp_tr.convert2string()), UVM_HIGH)
    VECT_CNT++;
    PASS_CNT++;
  endfunction

  function void ERROR(trans1 exp_tr, out_tr);
    `uvm_error("cmp vector",
       $sformatf("Actual %s does not match expected %s",
                 out_tr.output2string(),
                 exp_tr.convert2string()))
    VECT_CNT++;
    ERROR_CNT++;
  endfunction
endclass
```

Figure 34 - Example sb_comparator.sv - comparing transactions using out_tr.compare(exp_tr)

Implementing a proper `compare()` method using field macros or by overriding the `do_compare()` method in the transaction class greatly simplifies the creation of a UVM testbench.

8.12. uvm_comparer policy class methods

It should be noted that the `do_compare()` method has an often-overlooked second input argument of the `uvm_comparer` policy class with handle-name `comparer`.

Many examples in industry ignore the `comparer` handle and run the comparison calculations themselves and shown in the `do_compare()` method of Figure 35.

```
class trans9 extends uvm_sequence_item;
  rand bit [7:0] a, b, c;

  `uvm_object_utils_begin(trans9)
    `uvm_field_int(a, UVM_ALL_ON | UVM_NOCOMPARE)
    `uvm_field_int(b, UVM_ALL_ON | UVM_NOCOMPARE | UVM_NOCOPY)
    `uvm_field_int(c, UVM_ALL_ON | UVM_NOCOMPARE)
  `uvm_object_utils_end

  function new (string name="trans9");
    super.new(name);
  endfunction

  function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    trans9 tr;
    bit    eq;
    if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_compare() cast")
    eq  = super.do_compare(rhs, comparer);
    eq &= (a == tr.a);  // Compare outputs
    eq &= (b == tr.b);
    eq &= (c == tr.c);
    return(eq);
```

```
  endfunction

  `include "print_trans.sv"
endclass
```

Figure 35 - do_compare() method that does not use the uvm_comparer

An engineer may choose to take advantage of the built-in **uvm_comparer** methods to run the comparisons and print a standard output message when the compared fields do not match. The **trans10** class example in Figure 36 calls one of the **uvm_comparer** methods called **compare_field_int()**, with arguments that include a string name for the field being compared (for error reporting), the name of the local variable and the name of the compare-object variable along with the size of the variables being compared. As an interesting side note, the example of Figure 36 properly uses field macros to define most of the standard transaction methods but excludes the **compare()** method from field macro implementation. The **compare()** functionality is added by defining the **do_compare()** method in the **trans10** class (this technique was described at the beginning of Section 7).

```
class trans10 extends uvm_sequence_item;
  rand bit [7:0] a, b, c;

  `uvm_object_utils_begin(trans10)
    `uvm_field_int(a, UVM_ALL_ON | UVM_NOCOMPARE)
    `uvm_field_int(b, UVM_ALL_ON | UVM_NOCOMPARE | UVM_NOCOPY)
    `uvm_field_int(c, UVM_ALL_ON | UVM_NOCOMPARE)
  `uvm_object_utils_end

  function new (string name="trans10");
    super.new(name);
  endfunction

  function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    trans10 tr;
    bit    eq;
    if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_compare() cast")
    eq  = super.do_compare(rhs, comparer);
    eq &= comparer.compare_field_int("a", a, tr.a, 8);
    eq &= comparer.compare_field_int("b", b, tr.b, 8);
    eq &= comparer.compare_field_int("c", c, tr.c, 8);
    return(eq);
  endfunction

  `include "print_trans.sv"
endclass
```

Figure 36 - do_compare() method that DOES use the uvm_comparer methods

In both the **trans9** and **trans10** class examples, the **b**-variable was intentionally not copied to test the **do_compare()** methods and their accompanying error reporting capabilities. The **trans9** class example did user defined comparisons and the only reported error actually came from the top-level test. The **trans10** class example called the **comparer.compare_field_int()** methods, which did the comparisons and displayed additional **[MISCMP]** messages that are called from the built-in **compare_field_int()** method. The miscompare messages from both the **trans9** and **trans10** classes are shown in Figure 37.

```
// trans9 output
UVM_INFO body_run.sv(6) @ 0: uvm_test_top [tr1] inputs:a=be  b=93  c=44
UVM_INFO body_run.sv(7) @ 0: uvm_test_top [x1 ] inputs:a=be  b=00  c=44
UVM_ERROR body_run.sv(9) @ 0: uvm_test_top [ERRORCMP] x1 fields do NOT match tr1
fields

// trans10 output
UVM_INFO body_run.sv(6) @ 0: uvm_test_top [tr1] inputs:a=be  b=93  c=44
UVM_INFO body_run.sv(7) @ 0: uvm_test_top [x1 ] inputs:a=be  b=00  c=44
UVM_INFO @ 0: reporter [MISCMP] Miscompare for x1.b: lhs = 'h0 : rhs = 'h93
UVM_INFO @ 0: reporter [MISCMP] 1 Miscompare(s) for object tr1@464 vs. x1@468
UVM_ERROR body_run.sv(9) @ 0: uvm_test_top [ERRORCMP] x1 fields do NOT match tr1
fields
```

Figure 37 - Non-comparer output -vs- uvm_comparer reported messages

It is beyond the scope of this paper to go into detail regarding the **`uvm_comparer`** policy class, but there are a number of different knobs to control the **`uvm_comparer`** behavior, along with a number of built-in methods to help conduct comparisons. A short list of the built-in methods and an abbreviated description of their behavior as shown in the UVM Class Reference is shown in Table 1. The reader should reference the UVM Class Reference manual and review the **`uvm_comparer`** section.

compare field	Compares two integral values.
compare_field_int	This method is the same as compare field except that the arguments are small integers, less than or equal to 64 bits.
compare_field_real	This method is the same as compare field except that the arguments are real numbers.
compare_object	Compares two class objects using the policy knob to determine whether the comparison should be deep, shallow, or reference.
compare_string	Compares two string variables.
print_msg	Causes the error count to be incremented and the message, *msg*, to be appended to the miscompares string (a newline is used to separate messages).

Table 1 - uvm_comparer methods

8.13. do_methods & super.do_methods()

All of the empty, return-only **`do_methods()`** in the **`uvm_object`** base class mean that it is not necessary to ever call **`super.do_methods()`** from a transaction class that directly extends the **`uvm_sequence_item`**. The empty calls probably do no harm aside from potential minimal simulation efficiency issues related to calling empty void functions.

The default **`do_compare()`** method returns 1 because calls to **`super.do_compare()`** are typically **`and`**-ed with other comparison expressions, so if calling **`super.do_compare()`** returned empty or **`0`**-values, the compare method would always fail.

If the user-define transaction class is extended, then it becomes very important to call **`super.do_methods()`** to execute deep actions, such as deep-copy and deep-compare.

8.14. Templates with do_methods()

Coding the required **do_*methods()*** from scratch can be daunting, but large amounts of the **do_*methods()*** can be easily placed into a template file, which makes implementing the standard transaction methods using the **do_*methods()*** a relatively easy task. I use the **trans1** template file shown in Figure 38 as a starting point for my UVM testbench transactions.

```
class trans1 extends uvm_sequence_item;
  // (1)  Register class with factory      |
  `uvm_object_utils(trans1)

  // (2)  Declare variables & covergroups  | (if any)
       logic [15:0] dout;    // outputs not randomized
  rand bit   [15:0] din;
  rand bit          rst_n;

  // (5)  Standard new() constructor       |
  function new (string name="trans1");
    super.new(name);
  endfunction

  // (11) Common component & trans methods | (if any)
  function void do_copy(uvm_object rhs);
    trans1 tr;
    if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_copy() cast")
    // super.do_copy(rhs); // if extending an existing transaction
    // copy the transaction variables. Example:
    dout  = tr.dout;
    din   = tr.din;
    rst_n = tr.rst_n;
  endfunction

  function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    trans1 tr;
    bit    eq;
    if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_compare() cast")
    // super.do_compare(rhs, comparer); // if extending a transaction
    // compare the transaction output variables. Example:
    eq  = super.do_compare(rhs, comparer);
    eq &= (dout === tr.dout);
    return(eq);
  endfunction

  function void do_print(uvm_printer printer);
    $display("\n\n\t\t*** print() and sprint() are not implemented ",
             "for this transaction type ***\n\n");
  endfunction

  function string input2string();
    return($sformatf("din=%4h  rst_n=%b", din, rst_n));
  endfunction

  function string output2string();
    return($sformatf("dout=%4h", dout));
  endfunction
```

```
  function string convert2string();
    return($sformatf({input2string(), "   ", output2string()}));
  endfunction
endclass
```

Figure 38 - Example trans1.sv template file with do_copy() & do_compare() templates

All of the proper overhead declarations for the transaction handles and **`$cast`**ing have been captured in this **`trans1.sv`** template file, making it relatively easy to code the proper ***`do_methods()`*** for this transaction. This template happens to be a fully coded transaction class for a 16-bit, asynchronously resettable register.

9. Field macros

As previously mentioned and as shown in Figure 39, the second technique for defining all of the standard transaction methods is to declare the class data fields using field macros. Declaring the data fields using the built-in field macros is certainly easier to do than to redefine the ***do_methods()***, but the trade-off is simulation efficiency (see the section on Benchmarks for more details). The UVM Users Guide written by Cadence recommends the use of the field macros while Mentor developers discourage their use due to code expansion and simulation inefficiencies. Many users like to use the field macros because of their simplicity.

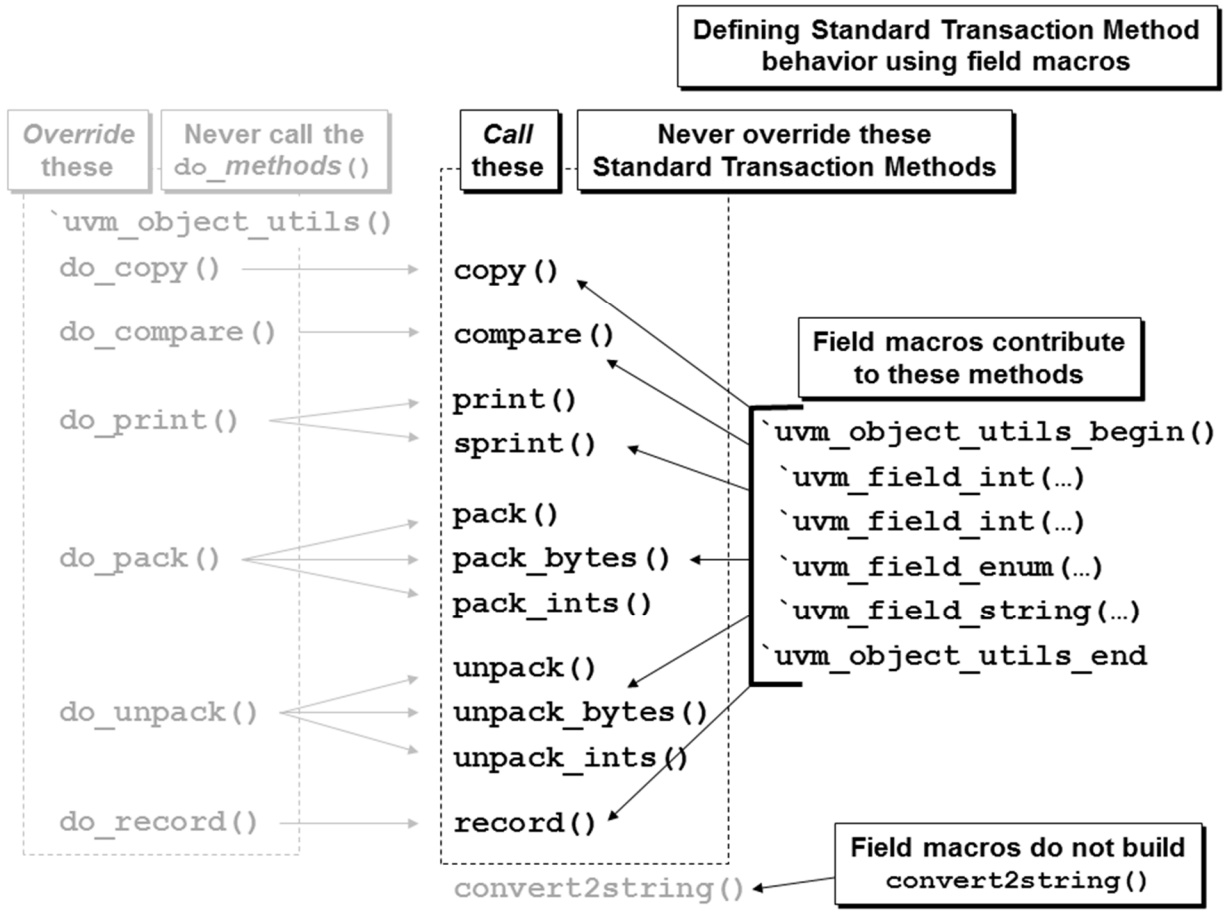

Figure 39 - Creating the standard transaction methods by using the UVM field macros

Rule: when using field macros, it is required to declare the transaction variables before they are specified in field macros.

Rule: when using field macros, the variables are declared before the registration of the transaction with the factory.

Rule: when using field macros, you must register the transaction with the factory using the **`uvm_object_utils_begin()** / **`uvm_object_utils_end** macros.

Note that even though variables can be declared in groups, as was done with the output variables **a-e** and the input variables **g-k** of Figure 40, the field macro declarations for these variables must include a unique **`uvm_field_int** declaration for each separate variable.

```
class trans1 extends uvm_sequence_item;
       bit [7:0] a, b, c, d, e; // outputs
  rand bit [2:0] g, h, i, j, k; //  inputs

  `uvm_object_utils_begin(trans1)
    `uvm_field_int(a, UVM_ALL_ON)
    `uvm_field_int(b, UVM_ALL_ON)
    `uvm_field_int(c, UVM_ALL_ON)
    `uvm_field_int(d, UVM_ALL_ON)
    `uvm_field_int(e, UVM_ALL_ON)
    `uvm_field_int(g, UVM_ALL_ON)
    `uvm_field_int(h, UVM_ALL_ON)
    `uvm_field_int(i, UVM_ALL_ON)
    `uvm_field_int(j, UVM_ALL_ON)
    `uvm_field_int(k, UVM_ALL_ON)
  `uvm_object_utils_end

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

Figure 40 - Creating the standard transaction methods by using the field macros

Trying to combine the variables into grouped field macro declarations as shown in Figure 41 causes a compilation error to occur (VCS error message shown at the bottom of Figure 41).

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

class trans1 extends uvm_sequence_item;
       bit [7:0] a, b, c, d, e; // outputs
  rand bit [2:0] g, h, i, j, k; //  inputs

  `uvm_object_utils_begin(trans1)
    `uvm_field_int(a, b, c, d, e, UVM_ALL_ON) // Error on this line
    `uvm_field_int(g, h, i, j, k, UVM_ALL_ON)
  `uvm_object_utils_end

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

// trans1_error2.sv, 9
```

```
//   Macro argument number mismatch for macro 'uvm_field_int'
//   "trans1_error2.sv", 9: token is ')'
//       `uvm_field_int(a, b, c, d, e, UVM_ALL_ON)
```

Figure 41 - ERROR - combining variables into a single field macro - VCS error shown

A logical follow-up question is, can we concatenate multiple variables into a single concatenated unit within a field macro declaration as shown in Figure 42? The answer is still no, and the resultant syntax error from VCS is also shown at the bottom of Figure 42.

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

class trans1 extends uvm_sequence_item;
       bit [7:0] a, b, c, d, e; // outputs
  rand bit [2:0] g, h, i, j, k; //  inputs

  `uvm_object_utils_begin(trans1)
    `uvm_field_int({a, b, c, d, a}, UVM_ALL_ON) // Error on this line
    `uvm_field_int({g, h, i, j, k}, UVM_ALL_ON)
  `uvm_object_utils_end

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

// Error-[SE] Syntax error
//   Following Verilog source has syntax error :
//   "trans1_error3.sv", 9 (expanding macro): token is '{'
//       `uvm_field_int({a, b, c, d, a}, UVM_ALL_ON) // Error on this line
```

Figure 42 - ERROR - concatenating variables into a single field macro - VCS error shown

Rule: when using field macros, each variable must be declared with a separate field macro. Variables cannot be grouped into a common field macro definition.

9.1. Field macro types

The most common field data type used in transactions is an integral numeric type (bits, vectors, buses, etc.), which requires declarations to be made with the **`uvm_field_int()** macro. There are certainly many other data types that can be used in a transaction.

To accommodate the multiple possible field types, UVM provides 35 field macros that can be used with the corresponding data types and all 35 have been defined in the file: **uvm/src/macros/uvm_sequence_defines.svh**.

Data declarations - field macro types		
`uvm_field_int	(ARG, FLAG)	Commonly used
`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)	
`uvm_field_sarray_int	(ARG, FLAG)	Static array
`uvm_field_sarray_enum	(ARG, FLAG)	
`uvm_field_sarray_object	(ARG, FLAG)	
`uvm_field_sarray_string	(ARG, FLAG)	
`uvm_field_array_int	(ARG, FLAG)	Dynamic array
`uvm_field_array_enum	(ARG, FLAG)	
`uvm_field_array_object	(ARG, FLAG)	
`uvm_field_array_string	(ARG, FLAG)	
`uvm_field_queue_int	(ARG, FLAG)	Queues
`uvm_field_queue_enum	(ARG, FLAG)	
`uvm_field_queue_object	(ARG, FLAG)	
`uvm_field_queue_string	(ARG, FLAG)	
`uvm_field_aa_string_int	(ARG, FLAG)	String assoc. array
`uvm_field_aa_string_string	(ARG, FLAG)	
`uvm_field_aa_object_int	(ARG, FLAG)	Class object assoc. array
`uvm_field_aa_object_string	(ARG, FLAG)	
`uvm_field_aa_int_int	(ARG, FLAG)	Number type assoc. array
`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)	

Table 2 - Field macros defined in UVM

I divide the 35 UVM field macros into seven categories as shown in Table 2:

- The first 6 are the most commonly used field macros.
- The next 4 are static array field macros.
- The next 4 are one-dimensional dynamic array field macros.
- The next 4 are queue field macros.
- The next 2 are string associative array field macros.
- The next 2 are class object field macros.
- The last 13 are integral number associative array field macros.

32 of the field macro take two arguments and three exception field macros require a third, leading key-type argument (**`uvm_field_enum** , **`uvm_field_aa_int_key** and **`uvm_field_aa_int_enumkey**). In Table 2, ARG is the name of the variable assigned to the field macro and FLAG specifies which standard transaction methods will be built for each field. As mentioned, enumerated type fields also require the corresponding T enumerated type, and integral-number associative arrays that are keyed to a specific type require the KEY key-type or enumerated-key-type.

9.2. Field macro flags

Field macro FLAG arguments are typically specified as either **UVM_ALL_ON** or **UVM_DEFAULT**, combined with flags that disable standard transaction method capabilities for specific variables.

On the former UVM World forum (now one of the forums on Accellera.org) I asked the UVM community which they preferred to use, **UVM_DEFAULT** or **UVM_ALL_ON** and why[10].

Two of the responses summarized prevailing opinions. From Kathleen Meade, UVM expert at Cadence:

> *My recommendation is to use UVM_DEFAULT instead of UVM_ALL_ON even though they both essentially do the same thing today. At some point the class library may add another "bit-flag" which may not necessarily be the DEFAULT. If you use UVM_ALL_ON that would imply that whatever flag it is would be "ON".*

A second and contrary opinion came from Ajeetha Kumari of CVC, India:

> *... we prefer ALL_ON to DEFAULT as it is more "explicit" in naming ... With DEFAULT - it is possible that a newer version of UVM base code might change the definition of default, and one (would need) to update the code!*

Both opinions expressed on the UVM forum are reasonable approaches, but in practice, I prefer to use the **UVM_ALL_ON** since I believe it better documents the action performed by this flag.

A related question is, what is the difference between **UVM_ALL_ON** and **UVM_DEFAULT**? To help answer this question, it is worth examining definitions from the

uvm/src/base/uvm_object_globals.svh source code file. There are two back-to-back sections in this file that shed some light and also introduce some confusion.

In the 23 lines of code shown in Table 3, it can be observed that there are 6 affirming field macro parameters (**UVM_DEFAULT** to **UVM_PACK**), 5 negating field macro parameters (**UVM_NOCOPY** to **UVM_NOPACK**), 3 depth and reference field macro parameters, and 1 more parameter called **UVM_READONLY**. Their supposed corresponding actions are shown with each parameter name.

```
// Parameter: `uvm_field_* macro flags
//
// Defines what operations a given field should be involved in.
// Bitwise OR all that apply.
//
// UVM_DEFAULT    - All field operations turned on
// UVM_COPY       - Field will participate in <uvm_object::copy>
// UVM_COMPARE    - Field will participate in <uvm_object::compare>
// UVM_PRINT      - Field will participate in <uvm_object::print>
// UVM_RECORD     - Field will participate in <uvm_object::record>
// UVM_PACK       - Field will participate in <uvm_object::pack>
//
// UVM_NOCOPY     - Field will not participate in <uvm_object::copy>
// UVM_NOCOMPARE  - Field will not participate in <uvm_object::compare>
// UVM_NOPRINT    - Field will not participate in <uvm_object::print>
// UVM_NORECORD   - Field will not participate in <uvm_object::record>
// UVM_NOPACK     - Field will not participate in <uvm_object::pack>
//
// UVM_DEEP       - Object field will be deep copied
// UVM_SHALLOW    - Object field will be shallow copied
// UVM_REFERENCE  - Object field will copied by reference
//
// UVM_READONLY   - Object field will NOT be automatically configured.
```

Table 3 - UVM field macro flag parameters defined in base/uvm_object_globals.svh

The Table 3 of parameter definitions does not appear in the UVM Reference Manual for three good reasons, (1) the list is incomplete (missing **UVM_ALL_ON**, **UVM_PHYSICAL**, **UVM_ABSTRACT**), (2) most of the affirming field macro parameters do nothing when put into user field macro definitions, and (3) the DEEP, SHALLOW and REFERENCE parameters are defined but commented out and hence are inactive, as shown in Table 4.

The field macro parameter values make up a 17-bit, onehot FLAG-vector. It can be seen in the Table 3 code that **UVM_DEFAULT** and **UVM_ALL_ON** both enable copy, compare, print, record and pack operations (all of these bits are hot), but **UVM_DEFAULT** also has the **UVM_DEEP** bit set. So will **UVM_DEFAULT** fields do deep-copies while **UVM_ALL_ON** fields only do shallow copies? The answer is no! There are more details about the various field macro flag settings following Figure 43.

```
parameter UVM_MACRO_NUMFLAGS    = 17;
//A=ABSTRACT Y=PHYSICAL
//F=REFERENCE, S=SHALLOW, D=DEEP
//K=PACK, R=RECORD, P=PRINT, M=COMPARE, C=COPY
//-------------------------- AYFSD K R P M C
```

```
parameter UVM_DEFAULT      = 'b000010101010101;
parameter UVM_ALL_ON       = 'b000000101010101;
parameter UVM_FLAGS_ON     = 'b000000101010101;
parameter UVM_FLAGS_OFF    = 0;

//Values are or'ed into a 32 bit value
//and externally
parameter UVM_COPY           = (1<<0);
parameter UVM_NOCOPY         = (1<<1);
parameter UVM_COMPARE        = (1<<2);
parameter UVM_NOCOMPARE      = (1<<3);
parameter UVM_PRINT          = (1<<4);
parameter UVM_NOPRINT        = (1<<5);
parameter UVM_RECORD         = (1<<6);
parameter UVM_NORECORD       = (1<<7);
parameter UVM_PACK           = (1<<8);
parameter UVM_NOPACK         = (1<<9);
//parameter UVM_DEEP           = (1<<10);
//parameter UVM_SHALLOW        = (1<<11);
//parameter UVM_REFERENCE      = (1<<12);
parameter UVM_PHYSICAL       = (1<<13);
parameter UVM_ABSTRACT       = (1<<14);
parameter UVM_READONLY       = (1<<15);
parameter UVM_NODEFPRINT     = (1<<16);
```

Table 4 - UVM field macro onehot flag settings in base/uvm_object_globals.svh

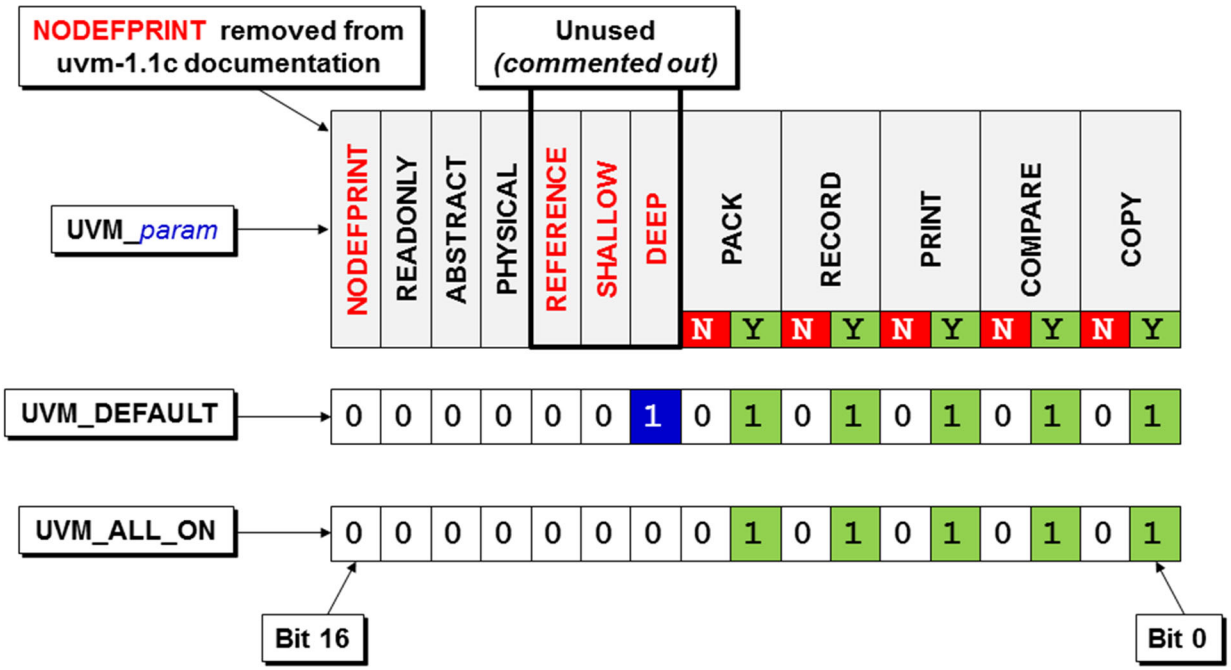

Figure 43- UVM field macro onehot flag settings diagram

From Table 4, it can be seen that **`UVM_DEEP`**, **`UVM_SHALLOW`** and **`UVM_REFERENCE`** are all commented out, and in practice. **`UVM_DEFAULT`** and **`UVM_ALL_ON`** both perform deep operations. Since there really are no active **`UVM_DEEP`** and **`UVM_SHALLON`** settings, **`UVM_DEFAULT`** and **`UVM_ALL_ON`** perform the exact same field operations.

The next interesting observation is that when **`UVM_ALL_ON`** is selected and combined with **`UVM_NOCOPY`**, both the copy-hot bit and the nocopy-hot bit are enabled, but the negating flag operations take precedence over the affirming flag operation.

There is one other interesting side effect from using negating flag settings. Quoting from the UVM Class Reference Manual, from the Field Macros section:

> *Each `uvm_field_* macro is named according to the particular data type it handles: integrals, strings, objects, queues, etc., and each has at least two arguments: ARG and FLAG.*
>
> *ARG* — *is the instance name of the variable, whose type must be compatible with the macro being invoked. ...*
>
> ***FLAG*** — *if set to UVM_ALL_ON, ... the ARG variable will be included in all data methods.* ***If FLAG is set to something other than UVM_ALL_ON or UVM_DEFAULT, it specifies which data method implementations will not include the given variable. Thus, if FLAG is specified as NO_COMPARE, the ARG variable will not affect comparison operations, but it will be included in everything else.***

The highlighted description for the ***FLAG*** argument leads to a rather surprising definition, which is that turning ***off*** one flag actually enables all other flags even without specifying **`UVM_ALL_ON`** or **`UVM_DEFAULT`**. The trans7 field macro definitions shown in Figure 44 actually enable **`UVM_ALL_ON`** and then disable **`pack()`** for the a variable, disable **`copy()`** for the **`b`**-variable and disable **`print()`** for the **`c`**-variable.

```
class trans7 extends uvm_sequence_item;
  rand bit [7:0] a, b, c;

  `uvm_object_utils_begin(trans7)
    `uvm_field_int(a, UVM_NOPACK)
    `uvm_field_int(b, UVM_NOCOPY)
    `uvm_field_int(c, UVM_NOPRINT)
  `uvm_object_utils_end

  function new (string name="trans7");
    super.new(name);
  endfunction

  `include "print_trans.sv"
endclass
```

Figure 44 - Field macro flags implicitly enable UVM_ALL_ON

In practice and for code clarity, engineers should specify either **`UVM_ALL_ON`** or **`UVM_DEFAULT`** followed by off-flags in an |-separated list.

Even though there are on-flags defined in the UVM class libraries, they do not appear to work as expected. Specifying **`UVM_COPY`** with no other flags actually turns on all of the other operations (**`copy()`**, **`compare()`**, **`print()`**, **`pack()`**, **`unpack()`**, **`record()`**).

The FLAG argument is frequently an **`or`**-separated list of flag settings but many industry examples use a **`+`**-separated list of flag settings as shown in the **`trans2`** definition of Figure 45

```
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_NOCOPY)
    `uvm_field_int(c, UVM_ALL_ON)
  `uvm_object_utils_end

  function new (string name="trans2");
    super.new(name);
  endfunction

  `include "print_trans.sv"
endclass
```

Figure 45 - trans2 legally defined using multiple +-separated field macro flags

When the **`trans2`** transaction is copied and compared using the **`test2`** code from Figure 46, the **`b`**-variable is intentionally not copied and the comparison for the **`b`**-variable fails, as can be seen in the simulation output shown in Figure 47.

```
class test2 extends uvm_test;
  `uvm_component_utils(test2)

  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  task run_phase(uvm_phase phase);
    trans2 tr1 = trans2::type_id::create("tr1");
    trans2 x1  = trans2::type_id::create("x1");
    //-------------------------------------
    phase.raise_objection(this);
    if (!tr1.randomize()) `uvm_fatal("FATALRAND", "tr1 Rand failed");
    x1.copy(tr1);
    $display("--------------\n\n");
    `uvm_info("tr1", tr1.convert2string(),UVM_MEDIUM);
    `uvm_info("x1 ",  x1.convert2string(),UVM_MEDIUM);
   if(x1.compare(tr1)) `uvm_info ("COMPARE" ,"x1 fields match tr1 fields",UVM_MEDIUM)
   else                `uvm_error("ERRORCMP","x1 fields do NOT match tr1 fields")
    $display("\n\n--------------");
    phase.drop_objection(this);
  endtask
endclass
```

Figure 46 - test2: copies and compares trans2 objects

```
UVM_INFO body_run.sv(6) @ 0: uvm_test_top [tr1] inputs:a=be  b=93  c=44
UVM_INFO body_run.sv(7) @ 0: uvm_test_top [x1 ] inputs:a=be  b=00  c=44
UVM_INFO @ 0: reporter [MISCMP] Miscompare for x1.b: lhs = 'h0 : rhs = 'h93
UVM_INFO @ 0: reporter [MISCMP] 1 Miscompare(s) for object tr1@464 vs. x1@468
UVM_ERROR body_run.sv(9) @ 0: uvm_test_top [ERRORCMP] x1 fields do NOT match tr1
fields
```

Figure 47 - test2 simulation output - b-variable comparison fails as expected

Selecting multiple flag settings with the |-separated list is a better option since it will not disable a desired action if a flag setting is accidentally used more than once as specified on the `b`-variable in Figure 48.

```
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_NOCOPY | UVM_ALL_ON | UVM_NOCOPY)
    `uvm_field_int(c, UVM_ALL_ON)
  `uvm_object_utils_end

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

  `include "print_trans.sv"
endclass
```

Figure 48 - UVM_NOCOPY flag accidentally |-specified twice - nocopy remains active

Selecting multiple flag settings with the +-separated list is subject to hot-bit clearing if a desired action flag setting is accidentally added more than once as specified on the `b`-variable in Figure 49.

```
class trans4 extends uvm_sequence_item;
  rand bit [7:0] a, b, c;

  `uvm_object_utils_begin(trans4)
    `uvm_field_int(a, UVM_ALL_ON)
    `uvm_field_int(b, UVM_NOCOPY + UVM_ALL_ON + UVM_NOCOPY) // Copies!!
    `uvm_field_int(c, UVM_ALL_ON)
  `uvm_object_utils_end

  function new (string name="trans4");
    super.new(name);
  endfunction

  `include "print_trans.sv"
endclass
```

Figure 49 - UVM_NOCOPY flag accidentally +-specified twice - removing the nocopy setting

Guideline: when using field macros, enable multiple flag settings using an |-separated list, not the +-separated list. This approach is safer if a flag setting is accidentally applied more than once.

9.3. Combining Field Macros with do_methods()

Is it possible to code some of the standard transaction methods partially using field macros and the other parts manually coded into **do_*methods()***? The short answer is yes, but the correct answer (to avoid confusion) is the two styles should not be mixed in the same transaction class definition.

The mixing of field macros and **do_*methods()*** would most likely occur if a base transaction class were defined using field macros and an extended transaction class were defined using **do_*methods()***. The **trans8b** base class uses field macros while the extended transaction class, **trans8**, overrides **do_copy()** and **do_compare()** methods as shown in Figure 50.

```
class trans8b extends uvm_sequence_item;
  rand bit [7:0] a, b;

  `uvm_object_utils_begin(trans8b)
    `uvm_field_int(a, UVM_ALL_ON)
    `uvm_field_int(b, UVM_ALL_ON | UVM_NOCOPY)
  `uvm_object_utils_end

  function new (string name="trans8b");
    super.new(name);
  endfunction
endclass

class trans8 extends trans8b;
  `uvm_object_utils(trans8)
  rand bit [7:0] c;

  function new (string name="trans8");
    super.new(name);
  endfunction

  function void do_copy(uvm_object rhs);
    trans8 tr;
    if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_copy() cast")
    super.do_copy(rhs);
    c = tr.c;
  endfunction

  function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    trans8 tr;
    bit    eq;
    if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_compare() cast")
    eq  = super.do_compare(rhs, comparer);
    eq &= (c == tr.c);  // Compare outputs
    return(eq);
  endfunction

  `include "print_trans.sv"
endclass
```

Figure 50 - trans8b base with field macros extended in trans8 with do_methods()

The **trans8** extended class inherits the **a** and **b** variables with field macros. Calling the **copy()** method for an extended **trans8** object first executes the copy operations for variables defined with field macros, then completes the **copy()** operation by calling the **do_copy()** method. The extended **trans8 do_copy()** method calls an empty inherited **do_copy()** method, which does nothing in the example in Figure 50.

Guideline: do not define field macros and override the corresponding ***do_methods()*** for the same standard transaction method in the same transaction class.

For instance, if a ***do_method()*** is defined for one of the standard transaction methods, then the method should be explicitly excluded from the field macros by setting the corresponding exclusion flag.

10. Benchmarks

Adam Erickson claimed that the ***do_methods()*** were more efficient both in code expansion and in simulation efficiency. Is that true?

I concede from Adam's paper that code expansion efficiency significantly favors implementation of the standard transaction methods using ***do_methods()***, but I decided to try running some benchmarks to prove or disprove Adam's claim about simulation efficiency.

Benchmarking can be tricky and needs to be specified in a way that can be repeated and give reasonable information. I ran the benchmarks using the latest simulators from two different vendors . Both simulators used built-in versions of UVM version 1.1d. The benchmark results will not report relative speeds between the vendor's simulators, since those numbers are highly dependent on the types of constructs used, but will report the relative percentage differences in simulation efficiency for each simulator when using different coding styles. The goal is to show users which coding styles will give the best results for all simulators.

10.1. Benchmarking methodology

The first benchmarks were run on a full UVM testbench environment with DUT but showed very little efficiency differences. Due to all the UVM activity within the full testbench environment, the efficiencies related to field macros versus ***do_methods()*** were largely masked. I then determined that I needed to isolate the standard transaction methods as much as possible, so the second set of benchmarks were done with just a test component with a tight loop that would repeatedly randomize, copy and compare transactions. The full top.sv, test.sv, tb_pkg files and transaction files are shown in the Appendix B.

The **run_phase()** test loop was run on one simulator using two different CNT values of 10-million and 100-million (I wanted to make sure that the efficiencies of the loop would not be overshadowed by startup and shutdown activities in the test). Then for comparison purposes, the

same code with CNT equal to 10-million was run on a second simulator. The **`run_phase()`** code of the test1 component is shown in Figure 51.

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

Figure 51 - Benchmark test1.sv run_phase() with randomize(), copy() and compare() loop

The transaction files were built using either **`do_methods()`** or field macros. Each transaction file included a common block of code as shown in Figure 52.

```
class trans1 extends uvm_sequence_item;

  // uvm_object_utils macro, data declarations
  // field macros if used

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

  // do_copy() & do_compare() methods if required

  function string input2string();
    return ($sformatf("g=%2h  h=%2h  i=%2h  j=%2h  k=%2h",
                      g,     h,     i,     j,     k));
  endfunction

  function string output2string();
    return ($sformatf("a=%2h  b=%2h  c=%2h  d=%2h  e=%2h",
                      a,     b,     c,     d,     e));
  endfunction

  function string convert2string();
    return ({"Inputs: ",   input2string(), "  ",
             "Outputs: ", output2string()});
  endfunction
endclass
```

Figure 52 - Common benchmark trans1 code

Each test was compiled once, and then run five times to gather data that could be averaged for comparison purposes. The script to compile and run the first **trans1** benchmark test is shown in Figure 53. Similar scripts exist for each **trans1**-coding benchmark variation.

```
vcs -sverilog -ntb_opts uvm -timescale=1ns/1ns -f run1a.f
/usr/bin/time -f "trans1a: no rand output - uses do_methods() - no field
macros  - simulation time %U" \
              -o log1a_1.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1a: no rand output - uses do_methods() - no field
macros  - simulation time %U" \
              -o log1a_2.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1a: no rand output - uses do_methods() - no field
macros  - simulation time %U" \
              -o log1a_3.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1a: no rand output - uses do_methods() - no field
macros  - simulation time %U" \
              -o log1a_4.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1a: no rand output - uses do_methods() - no field
macros  - simulation time %U" \
              -o log1a_5.vcs simv +UVM_TESTNAME=test1
cat log1a_*.vcs
```

Figure 53 - Benchmark script to run the first transactions five times

The script in Figure 53 shows string-text that wraps but in the actual script file the strings do not wrap.

10.2. Benchmarking do_methods() with nonrand-outputs and rand-outputs

Reminder: when implementing a transaction with the **do_*methods()*** the **`uvm_object_utils()** macro must be used.

The first transaction highlights, as shown in Figure 54, included:

(1) **`uvm_object_utils()** macro
(2) 5 non-**rand**, 8-bit, data outputs
(3) 5 **rand**, 8-bit, data inputs

do_copy() and **do_compare()** methods that called **super.*methods()***

The second benchmark transaction was identical to the first but also randomized the outputs. As mentioned earlier, there is no reason to randomize outputs since they will not be used. Will randomized outputs significantly impact simulation performance?

```
`uvm_object_utils(trans1)

     bit [7:0] a, b, c, d, e; // outputs
rand bit [2:0] g, h, i, j, k; //  inputs

function void do_copy(uvm_object rhs);
  trans1 tr;
  if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_copy() cast")
```

```
  super.do_copy(rhs);
  {a, b, c, d, e} = {tr.a, tr.b, tr.c, tr.d, tr.e};
  {g, h, i, j, k} = {tr.g, tr.h, tr.i, tr.j, tr.k};
endfunction

function bit do_compare(uvm_object rhs, uvm_comparer comparer);
  trans1 tr;
  bit    eq;
  if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_compare() cast")
  eq  = super.do_compare(rhs, comparer);
  eq &= (a == tr.a);
  eq &= (b == tr.b);
  eq &= (c == tr.c);
  eq &= (d == tr.d);
  eq &= (e == tr.e);
  eq &= (g == tr.g);
  eq &= (h == tr.h);
  eq &= (i == tr.i);
  eq &= (j == tr.j);
  eq &= (k == tr.k);
  return(eq);
endfunction
```

Figure 54 - First benchmark trans1 with non-rand outputs and do_methods()

Simulation results - needless randomization of the 5 output variables added simulation time as follows:

- Simulator A with CNT=10,000000: required 10.5% more simulation time
- Simulator A with CNT=100,000000: required 15.2% more simulation time
- Simulator B with CNT=10,000000: required 24.8% more simulation time

Clearly, one should not needlessly randomize variables that will not be used.

10.3. Benchmarking field macros with nonrand-outputs and rand-outputs

Reminder: when implementing a transaction with field macros the `` `uvm_object_utils_begin() `` / _end macros must be used.

The third transaction highlights, as shown in Figure 55, included:

(1) 5 non-**rand**, 8-bit, data outputs
(2) 5 **rand**, 8-bit, data inputs
(3) `` `uvm_object_utils_begin() `` macro
(4) `` `uvm_field_int `` macros with **UVM_ALL_ON**
(5) `` `uvm_object_utils_end ``

The fourth benchmark transaction was identical to the third but also randomized the outputs. As mentioned earlier, there is no reason to randomize outputs since they will not be used. Again, will randomized outputs significantly impact simulation performance?

```
     bit [7:0] a, b, c, d, e; // outputs
rand bit [2:0] g, h, i, j, k; //  inputs

`uvm_object_utils_begin(trans1)
  `uvm_field_int(a, UVM_ALL_ON)
  `uvm_field_int(b, UVM_ALL_ON)
  `uvm_field_int(c, UVM_ALL_ON)
  `uvm_field_int(d, UVM_ALL_ON)
  `uvm_field_int(e, UVM_ALL_ON)
  `uvm_field_int(g, UVM_ALL_ON)
  `uvm_field_int(h, UVM_ALL_ON)
  `uvm_field_int(i, UVM_ALL_ON)
  `uvm_field_int(j, UVM_ALL_ON)
  `uvm_field_int(k, UVM_ALL_ON)
`uvm_object_utils_end
```

Figure 55 - Third benchmark trans1 with non-rand outputs and field macros

Simulation results - needless randomization of the 5 output variables added simulation time as follows:

- Simulator A with CNT=10,000000: required 10.0% more simulation time
- Simulator A with CNT=100,000000: required 10.2% more simulation time
- Simulator B with CNT=10,000000: required 14.2% more simulation time

Clearly, one should not needlessly randomize variables that will not be used.

Benchmarking do_***methods()*** versus field macros

In addition to comparing rand versus non-rand outputs, simulation times were measured between **do_*method()*** and field macro versions to the **trans1** transactions (using the non-randomized outputs versions).

Simulation results - **do_*method()*** versions of the **trans1** transaction were more simulation efficient than the equivalent field macro versions of the **trans1** transaction. The added simulation time penalty for using the field macro versions were as follows:

- Simulator A with CNT=10,000000: required 4.5% more simulation time
- Simulator A with CNT=100,000000: required 6.4% more simulation time
- Simulator B with CNT=10,000000: required 94.7% more simulation time

As can be seen from the results, the **do_*method()*** version of the standard transaction methods is more simulation efficient than the equivalent field macro version. This is especially true using Simulator B where the measured penalty for using the field macros was 94.7% of additional simulation time.

Benchmarking do_***methods()*** versus do_***methods()*** without super.do_methods

As was previously mentioned, it is not necessary to call **super.do_copy()** and **super.do_compare()** for transactions that are extended from the **uvm_sequence_item** base

class. The reason was that the **do_copy()** and **do_compare()** methods in the base class were almost empty methods. So is there a penalty for calling the empty **super.do_copy()** and **super.do_compare()** methods? The answer is yes.

Simulation results - **do_*method()*** versions of the **trans1** transaction that did not call the **super.do_*methods*** were more simulation efficient than the equivalent transactions that called the **super.do_*methods***. The added simulation time penalty for calling the empty **super.do_*method()*** versions were as follows:

- Simulator A with CNT=10,000000: required 4.8% more simulation time
- Simulator A with CNT=100,000000: required 2.6% more simulation time
- Simulator B with CNT=10,000000: required 2.2% more simulation time

As can be seen from the results, calls to the empty **super.do_*method()*** versions is less simulation efficient than omitting the **super.do_*method()*** calls. Minor simulation speedups can be achieved by omitting the **super.do_*method()*** calls when they are unnecessary.

11. Summary & Conclusions

Classes are the preferred construct to represent transaction data because they are basically dynamic, ultra-flexible structs that can be easily randomized, easily control the randomization, and be created whenever they are needed.

The **uvm_sequence_item** and **int** class parameter types that are found in the UVM Base Class Library (BCL) are just placeholders that you will never use. Most of your testbench classes will be parameterized to the **trans1** (or name of your choice) transaction type, which is derived from the **uvm_sequence_item** type.

Using a standard class formatting style as shown in Figure 2, Figure 3 and Figure 4 makes it easier for users (and yourself) to understand and use your testbench component and transaction class implementations.

Rule: when using field macros, it is required to declare the transaction variables before they are specified in field macros.

Rule: when using field macros, the variables are declared before the registration of the transaction with the factory.

Rule: when using field macros, you must register the transaction with the factory using the **`uvm_object_utils_begin()** / **`uvm_object_utils_end** macros.

Rule: when using **do_*methods()***, you must register the transaction with the factory using the **`uvm_object_**utils () macro.

Rule: when using field macros, each variable must be declared with a separate field macro. Variables cannot be grouped into a common field macro definition.

Guideline: do not directly override the `copy()`, `compare()` and other `uvm_object` base class standard transaction methods.

Guideline: never manually implement the `create()` method. Call the `` `uvm_object_utils() `` macro to automatically implement the `create()` method.

Guideline: Every user-defined transaction method should include a `convert2string()` method.

Guideline: Avoid using the `print()` method. Its output is verbose and cannot be suppressed by using UVM verbosity settings.

Guideline: Avoid using the `sprint()` method. Its output is verbose.

Guideline: If you do use one of the built-in printing methods, choose `sprint()` over `print()` and call it from a UVM message macro. Runtime verbosity settings can mask verbose `sprint()` method printouts if desired.

Guideline: Define and use the `convert2string()` method discussed in earlier sections. `convert2string()` is more simulation efficient, more print-space efficient and can be easily suppressed by using different runtime UVM verbosity settings.

There are additional guidelines included throughout the paper, but following these rules and guidelines are the current Best Known Methods for using UVM transactions.

12. Acknowledgements

I am grateful to my colleague Stuart Sutherland for his exhaustive review of this paper, for identifying errors and suggesting improvements to the content and flow of this paper. I am also grateful to my colleague Heath Chambers for identifying sections that could be merged to improve the flow of this paper.

13. References:

[1] Adam Erickson, "Are OVM & UVM Macros Evil? A Cost-Benefit Analysis," DVCon 2011. Copy can also be requested at: http://www.mentor.com/products/fv/verificationhorizons/horizons-jun-11

[2] Clifford E. Cummings, "OVM/UVM Scoreboards - Fundamental Architectures," SNUG-SV 2013 - www.sunburst-design.com/papers/CummingsSNUG2013SV_UVM_Scoreboards.pdf

[3] Dave Rich, Tom Fitzpatrick - Mentor UVM Experts - personal communication

[4] Kathleen A. Meade, Sharon Rosenberg, A Practical Guide to Adopting the Universal Verification Methodology (UVM), Second Edition, ISBN 978-1-300-53593-5. Published 2013

[5] Kathleen A. Meade, Sharon Rosenberg - Cadence UVM Experts - personal communication

[6] OVM User Guide, March 2010, Available for download from: https://verificationacademy.com/forums/downloads/ovm/uvm-download-kits

[7] Universal Verification Methodology (UVM) 1.1 Class Reference, May 2011, Accellera, Napa, CA. www.accellera.org/downloads/standards/uvm

[8] Universal Verification Methodology (UVM) 1.1 Users Guide, May 18, 2011, Accellera, Napa, CA. www.accellera.org/downloads/standards/uvm

[9] UVM (Universal Verification Methodology) Forum - Accellera Systems Initiative Forums, http://forums.accellera.org/topic/991-uvm-all-on-vs-uvm-default/?hl=uvm_default

[10] UVM_DEFAULT -vs- UVM_ALL_ON, http://forums.accellera.org/topic/991-uvm-all-on-vs-uvm-default/?hl=uvm_default

[11] Vanessa R. Cooper, Getting Started with UVM: A Beginner's Guide, ISBN-10: 0615819974 | ISBN-13: 978-0615819976, Published 2013

[12] Verification Academy, https://verificationacademy.com/

14. AUTHOR & CONTACT INFORMATION

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 32 years of ASIC, FPGA and system design experience and 23 years of SystemVerilog, synthesis and methodology training experience.

Mr Cummings has presented more than 100 SystemVerilog seminars and training classes in the past nine years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Mr Cummings has participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee, and has presented more than 40 papers on SystemVerilog & SystemVerilog related design, synthesis and verification techniques.

Mr Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

Sunburst Design, Inc. offers World Class Verilog & SystemVerilog training courses. For more information, visit the www.sunburst-design.com web site.
Email address: cliffc@sunburst-design.com

Last Updated: March 31, 2014

15. Appendix A

15.1. UVM classes parameterized to uvm_sequence_item

Many of the UVM base classes are parameterized classes, also known as specializations of classes. UVM version 1.1d includes eight base classes that are parameterized to the **`uvm_sequence_item`** type as shown in Figure 56. Seven of the base classes are component classes and the eighth is the sequence base class.

```
File: comps/uvm_driver.svh
  uvm_driver                  #(type REQ = uvm_sequence_item, ...)

File: comps/uvm_push_driver.svh
  uvm_push_driver             #(type REQ = uvm_sequence_item, ...)

File: seq/uvm_push_sequencer.svh
  uvm_push_sequencer          #(type REQ = uvm_sequence_item, ...)

File: seq/uvm_sequence.svh
  uvm_sequence                #(type REQ = uvm_sequence_item, ...)

File: seq/uvm_sequence_library.svh
  uvm_sequence_library        #(type REQ = uvm_sequence_item, ...)

File: seq/uvm_sequencer.svh
  uvm_sequencer               #(type REQ = uvm_sequence_item, ...)

File: seq/uvm_sequencer_analysis_fifo.svh
  uvm_sequencer_analysis_fifo #(type RSP = uvm_sequence_item     )

File: seq/uvm_sequencer_param_base.svh
  uvm_sequencer_param_base    #(type REQ = uvm_sequence_item, ...)
```

Figure 56 - UVM classes parameterized to the uvm_sequence_item type

15.2. UVM classes parameterized to int

Many of the UVM base classes are parameterized classes, also known as specializations of classes. UVM version 1.1d includes 69 base classes that are parameterized to the **`int`** type as shown in Figure 57.

```
File: base/uvm_config_db.svh
  uvm_config_db                    #(type T=int)

File: base/uvm_queue.svh
  uvm_queue                        #(type T=int)

File: base/uvm_resource.svh
  uvm_resource                     #(type T=int)

File: base/uvm_spell_chkr.svh
  uvm_spell_chkr                   #(type T=int)
```

```
File: comps/uvm_in_order_comparator.svh
  uvm_in_order_built_in_comparator  #(type T=int)
  uvm_in_order_class_comparator     #(type T=int)

File: comps/uvm_policies.svh
  uvm_built_in_comp                 #(type T=int)
  uvm_built_in_converter            #(type T=int)
  uvm_built_in_clone                #(type T=int)
  uvm_class_comp                    #(type T=int)
  uvm_class_converter               #(type T=int)
  uvm_class_clone                   #(type T=int)

File: comps/uvm_subscriber.svh
  uvm_subscriber                    #(type T=int)

File: macros/uvm_tlm_defines.svh
  uvm_blocking_put_imp``SFX         #(type T=int, type IMP=int)
  uvm_nonblocking_put_imp``SFX      #(type T=int, type IMP=int)
  uvm_put_imp``SFX                  #(type T=int, type IMP=int)
  uvm_blocking_get_imp``SFX         #(type T=int, type IMP=int)
  uvm_nonblocking_get_imp``SFX      #(type T=int, type IMP=int)
  uvm_get_imp``SFX                  #(type T=int, type IMP=int)
  uvm_blocking_peek_imp``SFX        #(type T=int, type IMP=int)
  uvm_nonblocking_peek_imp``SFX     #(type T=int, type IMP=int)
  uvm_peek_imp``SFX                 #(type T=int, type IMP=int)
  uvm_blocking_get_peek_imp``SFX    #(type T=int, type IMP=int)
  uvm_nonblocking_get_peek_imp``SFX #(type T=int, type IMP=int)
  uvm_get_peek_imp``SFX             #(type T=int, type IMP=int)
  uvm_analysis_imp``SFX             #(type T=int, type IMP=int)

File: tlm1/uvm_analysis_port.svh
  uvm_analysis_imp                  #(type T=int, type IMP=int)
  uvm_analysis_export               #(type T=int)
  uvm_analysis_port                 #(type T=int)

File: tlm1/uvm_exports.svh
  uvm_blocking_put_export           #(type T=int)
  uvm_nonblocking_put_export        #(type T=int)
  uvm_put_export                    #(type T=int)
  uvm_blocking_get_export           #(type T=int)
  uvm_nonblocking_get_export        #(type T=int)
  uvm_get_export                    #(type T=int)
  uvm_blocking_peek_export          #(type T=int)
  uvm_nonblocking_peek_export       #(type T=int)
  uvm_peek_export                   #(type T=int)
  uvm_blocking_get_peek_export      #(type T=int)
  uvm_nonblocking_get_peek_export   #(type T=int)
  uvm_get_peek_export               #(type T=int)

File: tlm1/uvm_imps.svh
  uvm_blocking_put_imp              #(type T=int, type IMP=int)
  uvm_nonblocking_put_imp           #(type T=int, type IMP=int)
  uvm_put_imp                       #(type T=int, type IMP=int)
  uvm_blocking_get_imp              #(type T=int, type IMP=int)
  uvm_nonblocking_get_imp           #(type T=int, type IMP=int)
  uvm_get_imp                       #(type T=int, type IMP=int)
```

```
  uvm_blocking_peek_imp             #(type T=int, type IMP=int)
  uvm_nonblocking_peek_imp          #(type T=int, type IMP=int)
  uvm_peek_imp                      #(type T=int, type IMP=int)
  uvm_blocking_get_peek_imp         #(type T=int, type IMP=int)
  uvm_nonblocking_get_peek_imp      #(type T=int, type IMP=int)
  uvm_get_peek_imp                  #(type T=int, type IMP=int)

File: tlm1/uvm_ports.svh
  uvm_blocking_put_port             #(type T=int)
  uvm_nonblocking_put_port          #(type T=int)
  uvm_put_port                      #(type T=int)
  uvm_blocking_get_port             #(type T=int)
  uvm_nonblocking_get_port          #(type T=int)
  uvm_get_port                      #(type T=int)
  uvm_blocking_peek_port            #(type T=int)
  uvm_nonblocking_peek_port         #(type T=int)
  uvm_peek_port                     #(type T=int)
  uvm_blocking_get_peek_port        #(type T=int)
  uvm_nonblocking_get_peek_port     #(type T=int)
  uvm_get_peek_port                 #(type T=int)

File: tlm1/uvm_tlm_fifo_base.svh
  uvm_tlm_fifo_base                 #(type T=int)

File: tlm1/uvm_tlm_fifos.svh
  uvm_tlm_analysis_fifo             #(type T=int)
  uvm_tlm_fifo                      #(type T=int)

File: tlm2/uvm_tlm2_generic_payload.svh
  uvm_tlm_extension                 #(type T=int)
```

Figure 57 - UVM classes parameterized to the int type

16. Appendix B

16.1. Benchmark files to test simulation efficiency

This section contains the files that were used to run the simulations referenced in the Benchmarks section. There were six different `trans1` transaction coding styles tested. Example 1- Example 12 are the `tb_pkg1[a-f].sv` files and `run1[a-f].f` files used by the benchmark simulations.

```
`include "uvm_macros.svh"
package tb_pkg;
  import uvm_pkg::*;
  `include "trans1a.sv"
  `include "test1.sv"
endpackage
```

Example 1 - File: tb_pkg1a.sv

```
tb_pkg1a.sv
top.sv
```

Example 2 - File: run1a.f

```
`include "uvm_macros.svh"
package tb_pkg;
  import uvm_pkg::*;
  `include "trans1b.sv"
  `include "test1.sv"
endpackage
```

Example 3 - File: tb_pkg1b.sv

```
tb_pkg1b.sv
top.sv
```

Example 4 - File: run1b.f

```
`include "uvm_macros.svh"
package tb_pkg;
  import uvm_pkg::*;
  `include "trans1c.sv"
  `include "test1.sv"
endpackage
```

Example 5 - File: tb_pkg1c.sv

```
tb_pkg1c.sv
top.sv
```

Example 6 - File: run1c.f

```
`include "uvm_macros.svh"
package tb_pkg;
  import uvm_pkg::*;
  `include "trans1d.sv"
  `include "test1.sv"
endpackage
```

Example 7 - File: tb_pkg1d.sv

```
tb_pkg1d.sv
top.sv
```

Example 8 - File: run1d.f

```
`include "uvm_macros.svh"
package tb_pkg;
  import uvm_pkg::*;
  `include "trans1e.sv"
  `include "test1.sv"
endpackage
```

Example 9 - File: tb_pkg1e.sv

```
tb_pkg1e.sv
top.sv
```

Example 10 - File: run1e.f

```
`include "uvm_macros.svh"
package tb_pkg;
  import uvm_pkg::*;
  `include "trans1f.sv"
  `include "test1.sv"
endpackage
```

Example 11 - File: tb_pkg1f.sv

```
tb_pkg1f.sv
top.sv
```

Example 12 - File: run1f.f

Each benchmarked transaction was first compiled and then simulated five times. The average of the five simulation runs were compared to other transaction simulations.

```
vcs -sverilog -ntb_opts uvm -timescale=1ns/1ns -f run1a.f
/usr/bin/time -f "trans1a: no rand output - uses do_methods() - no field macros  - simulation time %U" \
              -o log1a_1.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1a: no rand output - uses do_methods() - no field macros  - simulation time %U" \
              -o log1a_2.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1a: no rand output - uses do_methods() - no field macros  - simulation time %U" \
              -o log1a_3.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1a: no rand output - uses do_methods() - no field macros  - simulation time %U" \
              -o log1a_4.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1a: no rand output - uses do_methods() - no field macros  - simulation time %U" \
              -o log1a_5.vcs simv +UVM_TESTNAME=test1
cat log1a*.vcs
```

Example 13 - File: doit1a.vcs

```
vcs -sverilog -ntb_opts uvm -timescale=1ns/1ns -f run1b.f
/usr/bin/time -f "trans1b:    rand output - uses do_methods() - no field macros  - simulation time %U" \
              -o log1b_1.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1b:    rand output - uses do_methods() - no field macros  - simulation time %U" \
              -o log1b_2.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1b:    rand output - uses do_methods() - no field macros  - simulation time %U" \
              -o log1b_3.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1b:    rand output - uses do_methods() - no field macros  - simulation time %U" \
              -o log1b_4.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1b:    rand output - uses do_methods() - no field macros  - simulation time %U" \
              -o log1b_5.vcs simv +UVM_TESTNAME=test1
cat log1b*.vcs
```

Example 14- File: doit1b.vcs

```
vcs -sverilog -ntb_opts uvm -timescale=1ns/1ns -f run1c.f
/usr/bin/time -f "trans1c: no rand output - uses field macros - no do_methods()  - simulation time %U" \
              -o log1c_1.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1c: no rand output - uses field macros - no do_methods()  - simulation time %U" \
              -o log1c_2.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1c: no rand output - uses field macros - no do_methods()  - simulation time %U" \
              -o log1c_3.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1c: no rand output - uses field macros - no do_methods()  - simulation time %U" \
              -o log1c_4.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1c: no rand output - uses field macros - no do_methods()  - simulation time %U" \
              -o log1c_5.vcs simv +UVM_TESTNAME=test1
cat log1c*.vcs
```

Example 15- File: doit1c.vcs

```
vcs -sverilog -ntb_opts uvm -timescale=1ns/1ns -f run1d.f
/usr/bin/time -f "trans1d:    rand output - uses filed macros - no do_methods()  - simulation time %U" \
              -o log1d_1.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1d:    rand output - uses filed macros - no do_methods()  - simulation time %U" \
              -o log1d_2.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1d:    rand output - uses filed macros - no do_methods()  - simulation time %U" \
              -o log1d_3.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1d:    rand output - uses filed macros - no do_methods()  - simulation time %U" \
              -o log1d_4.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1d:    rand output - uses filed macros - no do_methods()  - simulation time %U" \
              -o log1d_5.vcs simv +UVM_TESTNAME=test1
cat log1d*.vcs
```

Example 16- File: doit1d.vcs

```
vcs -sverilog -ntb_opts uvm -timescale=1ns/1ns -f run1e.f
/usr/bin/time -f "trans1e: no rand output - do_methods() - no super.do_methods() - simulation time %U" \
              -o log1e_1.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1e: no rand output - do_methods() - no super.do_methods() - simulation time %U" \
              -o log1e_2.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1e: no rand output - do_methods() - no super.do_methods() - simulation time %U" \
              -o log1e_3.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1e: no rand output - do_methods() - no super.do_methods() - simulation time %U" \
              -o log1e_4.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1e: no rand output - do_methods() - no super.do_methods() - simulation time %U" \
              -o log1e_5.vcs simv +UVM_TESTNAME=test1
cat log1e*.vcs
```

Example 17- File: doit1e.vcs

```
vcs -sverilog -ntb_opts uvm -timescale=1ns/1ns -f run1f.f
/usr/bin/time -f "trans1f: no rand output - UVM_NOPACK, UVM_NOCOMPARE removed UVM_ALL_ON field macros - simtime %U" \
              -o log1f_1.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1f: no rand output - UVM_NOPACK, UVM_NOCOMPARE removed UVM_ALL_ON field macros - simtime %U" \
              -o log1f_2.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1f: no rand output - UVM_NOPACK, UVM_NOCOMPARE removed UVM_ALL_ON field macros - simtime %U" \
              -o log1f_3.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1f: no rand output - UVM_NOPACK, UVM_NOCOMPARE removed UVM_ALL_ON field macros - simtime %U" \
              -o log1f_4.vcs simv +UVM_TESTNAME=test1
/usr/bin/time -f "trans1f: no rand output - UVM_NOPACK, UVM_NOCOMPARE removed UVM_ALL_ON field macros - simtime %U" \
              -o log1f_5.vcs simv +UVM_TESTNAME=test1
cat log1f*.vcs
```

Example 18 - File: doit1f.vcs

The **report.vcs** file concatenates all of the benchmark simulation results into a single file called **vcs_benchmark_times**, and then moves all of the separate benchmark report files into a **VCSLOG** directory, along with a copy of the **vcs_benchmark_times** file.

```
rm -rf vcs_benchmark_times
cat log1*.vcs > vcs_benchmark_times
mv log1*.vcs VCSLOG
cp -rp vcs_benchmark_times VCSLOG
```

Example 19 - File: report.vcs - gathers benchmark simulation times

The script to start the benchmark simulations is the **doitall.vcs** script. This script should be executed after setting the repeat-loop count value (**CNT**) in the file: **CNT_file**

```
doit1a.vcs
doit1b.vcs
doit1c.vcs
doit1d.vcs
doit1e.vcs
doit1f.vcs
report.vcs
```

Example 20 - File: doitall.vcs - execute after setting loop CNT value in the CNT_file file

Each of the **trans1** class examples **`includes** a common set of printing methods. Including the print-methods reduces the code volume for each of the **trans1** class examples.

```
  function string input2string();
    return ($sformatf("g=%2h  h=%2h  i=%2h  j=%2h  k=%2h",
                      g,     h,     i,     j,     k));
  endfunction

  function string output2string();
    return ($sformatf("a=%2h  b=%2h  c=%2h  d=%2h  e=%2h",
                      a,     b,     c,     d,     e));
  endfunction

  function string convert2string();
    return ({"Inputs: ",   input2string(), "  ",
             "Outputs: ", output2string()});
  endfunction
```

Example 21 - trans_printing.sv - common printing methods included in each trans1 class

The **top.sv** file is top-module used to run the simulations.

```
`include "uvm_macros.svh"
module top;
  import uvm_pkg::*; // import uvm base  classes
  import  tb_pkg::*; // import testbench classes

  initial run_test();
endmodule
```

Example 22 - File: top.sv - wrapper top-module to permit testing

The **test1** loop **CNT** value used by the **test1** class shown in Example 24 is controlled by changing the **CNT** (**repeat**-loop limit) value in the **CNT_file**, which is **`included** into the **test1.sv** file.

```
`define CNT 10_000_000
```

Example 23 - File: CNT_file - holds loop-CNT value

16.2. Benchmark vcs_benchmark_times file

The actual benchmark output file for running the VCS benchmarks with a loop **`CNT** = 10 million is shown below. There are five results for each **trans1** transaction type. This file was generated by executing the **doitall.vcs** script.

```
trans1a: no rand output - uses do_methods() - no field macros  - simulation time 126.07
trans1a: no rand output - uses do_methods() - no field macros  - simulation time 128.66
trans1a: no rand output - uses do_methods() - no field macros  - simulation time 125.69
trans1a: no rand output - uses do_methods() - no field macros  - simulation time 129.05
trans1a: no rand output - uses do_methods() - no field macros  - simulation time 124.73
trans1b:    rand output - uses do_methods() - no field macros  - simulation time 138.95
trans1b:    rand output - uses do_methods() - no field macros  - simulation time 139.17
trans1b:    rand output - uses do_methods() - no field macros  - simulation time 141.62
trans1b:    rand output - uses do_methods() - no field macros  - simulation time 139.50
trans1b:    rand output - uses do_methods() - no field macros  - simulation time 141.79
trans1c: no rand output - uses field macros - no do_methods()  - simulation time 132.95
trans1c: no rand output - uses field macros - no do_methods()  - simulation time 127.93
trans1c: no rand output - uses field macros - no do_methods()  - simulation time 134.30
trans1c: no rand output - uses field macros - no do_methods()  - simulation time 131.59
trans1c: no rand output - uses field macros - no do_methods()  - simulation time 135.68
trans1d:    rand output - uses filed macros - no do_methods()  - simulation time 144.47
trans1d:    rand output - uses filed macros - no do_methods()  - simulation time 151.20
trans1d:    rand output - uses filed macros - no do_methods()  - simulation time 145.12
trans1d:    rand output - uses filed macros - no do_methods()  - simulation time 144.39
trans1d:    rand output - uses filed macros - no do_methods()  - simulation time 143.90
trans1e: no rand output - do_methods() - no super.do_methods() - simulation time 122.51
trans1e: no rand output - do_methods() - no super.do_methods() - simulation time 121.24
trans1e: no rand output - do_methods() - no super.do_methods() - simulation time 119.36
trans1e: no rand output - do_methods() - no super.do_methods() - simulation time 120.73
trans1e: no rand output - do_methods() - no super.do_methods() - simulation time 120.12
trans1f: no rand output - UVM_NOPACK, UVM_NOCOMPARE removed UVM_ALL_ON field macros - simtime 134.00
trans1f: no rand output - UVM_NOPACK, UVM_NOCOMPARE removed UVM_ALL_ON field macros - simtime 128.78
trans1f: no rand output - UVM_NOPACK, UVM_NOCOMPARE removed UVM_ALL_ON field macros - simtime 130.62
trans1f: no rand output - UVM_NOPACK, UVM_NOCOMPARE removed UVM_ALL_ON field macros - simtime 133.32
trans1f: no rand output - UVM_NOPACK, UVM_NOCOMPARE removed UVM_ALL_ON field macros - simtime 133.64
```

Figure 58 - vcs_benchmark_times report file for a loop CNT=10,000,000

16.3. Benchmark test1 file with repeat-loop

```
`include "CNT_file"

class test1 extends uvm_test;
  `uvm_component_utils(test1)
  int VECT_CNT, PASS_CNT, ERROR_CNT;
  string pstr = "\n\n\n*** TEST PASSED - ";
  string estr = "\n\n\n*** TEST FAILED - ";

  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  task run_phase(uvm_phase phase);
    trans1 tr1 = trans1::type_id::create("tr1");
    trans1 x1  = trans1::type_id::create("x1");
    //--------------------------------------
    phase.raise_objection(this);
    $display("--------------\n\n");
    repeat(`CNT) begin
      if (!tr1.randomize()) `uvm_fatal("FATALRAND", "tr1 rand failed");
      x1.copy(tr1);
      `uvm_info("tr1", tr1.convert2string(),UVM_DEBUG);
      `uvm_info(" x1", x1.convert2string(), UVM_DEBUG);
      if (x1.compare(tr1)) PASS (tr1);
      else                 ERROR(tr1, x1);
    end
    $display("\n\n--------------");
    phase.drop_objection(this);
  endtask

  function void report_phase(uvm_phase phase);
    if (VECT_CNT && !ERROR_CNT) `uvm_info ("TEST PASSED",
      $sformatf({pstr, "vectors: %0d ran, %0d passed ***\n"},
                VECT_CNT, PASS_CNT), UVM_LOW)
    else                        `uvm_error("TEST FAILED",
      $sformatf({estr, "vectors: %0d ran, %0d passed , %0d failed ***\n"},
                VECT_CNT, PASS_CNT, ERROR_CNT))
  endfunction

  function void PASS(trans1 exp_tr);
    `uvm_info("PASSMSG",
       $sformatf("Vec#%0d:\n\tPassed: %s",
                 VECT_CNT, exp_tr.convert2string()), UVM_HIGH)
    VECT_CNT++;
    PASS_CNT++;
  endfunction

  function void ERROR(trans1 exp_tr, out_tr);
    `uvm_error("ERRORMSG",
       $sformatf("Vec#%0d:\n\tActual: %s\n\tExpect: %s",
                 VECT_CNT, out_tr.convert2string(),
                           exp_tr.convert2string()))
    VECT_CNT++;
    ERROR_CNT++;
  endfunction
endclass
```

Example 24 - File: test1.sv - randomizes, copies and compares in a repeat(`CNT) loop

The **test1** class, shown in Example 24, has a **run_phase()** that factory-creates two class objects, **tr1** and **x1**, and goes into a **repeat** loop that randomizes the **tr1** variables, copies the **tr1** variables to the **x1** variables, and then compares the values of the **tr1** variables to the **x1** variables. This is a tight loop that is repeated millions of times to benchmark performance differences related to how the **copy()** and **compare()** methods were created in different **trans1** classes.

trans1a - non-randomized outputs - do_***methods()*** - no field macros

The **trans1** class defined in the **trans1a.sv** file has five non-randomized outputs and five randomized inputs. The **trans1a.sv** example has user-defined **do_copy()** and **do_compare()** methods but no field macro definitions.

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

       bit [7:0] a, b, c, d, e; // outputs
  rand bit [2:0] g, h, i, j, k; //  inputs

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

  function void do_copy(uvm_object rhs);
    trans1 tr;
    if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_copy() cast")
    super.do_copy(rhs);
    {a, b, c, d, e} = {tr.a, tr.b, tr.c, tr.d, tr.e};
    {g, h, i, j, k} = {tr.g, tr.h, tr.i, tr.j, tr.k};
  endfunction

  function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    trans1 tr;
    bit    eq;
    if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_compare() cast")
    eq  = super.do_compare(rhs, comparer);
    eq &= (a == tr.a);  // Compare outputs
    eq &= (b == tr.b);
    eq &= (c == tr.c);
    eq &= (d == tr.d);
    eq &= (e == tr.e);
    eq &= (g == tr.g);
    eq &= (h == tr.h);
    eq &= (i == tr.i);
    eq &= (j == tr.j);
    eq &= (k == tr.k);
    return(eq);
  endfunction

  `include "trans_printing.sv"
endclass
```

Example 25 - File: trans1a.sv - no rand outputs - uses do_methods() - no field macros

trans1b - randomized outputs - do_***methods()*** - no field macros

The **trans1** class defined in the **trans1b.sv** file has five randomized outputs and five randomized inputs. The **trans1b.sv** example has user-defined **do_copy()** and **do_compare()** methods but no field macro definitions.

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

  rand bit [7:0] a, b, c, d, e; // outputs
  rand bit [2:0] g, h, i, j, k; //  inputs

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

  function void do_copy(uvm_object rhs);
    trans1 tr;
    if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_copy() cast")
    super.do_copy(rhs);
    {a, b, c, d, e} = {tr.a, tr.b, tr.c, tr.d, tr.e};
    {g, h, i, j, k} = {tr.g, tr.h, tr.i, tr.j, tr.k};
  endfunction

  function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    trans1 tr;
    bit    eq;
    if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_compare() cast")
    eq  = super.do_compare(rhs, comparer);
    eq &= (a == tr.a);  // Compare outputs
    eq &= (b == tr.b);
    eq &= (c == tr.c);
    eq &= (d == tr.d);
    eq &= (e == tr.e);
    eq &= (g == tr.g);
    eq &= (h == tr.h);
    eq &= (i == tr.i);
    eq &= (j == tr.j);
    eq &= (k == tr.k);
    return(eq);
  endfunction

  `include "trans_printing.sv"
endclass
```

Example 26- File: trans1b.sv - rand outputs - uses do_methods() - no field macros

trans1c - no randomized outputs - uses field macros - no ***do_methods()***

The **trans1** class defined in the **trans1c.sv** file has five non-randomized outputs and five randomized inputs. The **trans1c.sv** example has user-defined field macro definitions but no ***do_methods()***.

```
class trans1 extends uvm_sequence_item;
       bit [7:0] a, b, c, d, e; // outputs
  rand bit [2:0] g, h, i, j, k; //  inputs

  `uvm_object_utils_begin(trans1)
    `uvm_field_int(a, UVM_ALL_ON)
    `uvm_field_int(b, UVM_ALL_ON)
    `uvm_field_int(c, UVM_ALL_ON)
    `uvm_field_int(d, UVM_ALL_ON)
    `uvm_field_int(e, UVM_ALL_ON)
    `uvm_field_int(g, UVM_ALL_ON)
    `uvm_field_int(h, UVM_ALL_ON)
    `uvm_field_int(i, UVM_ALL_ON)
    `uvm_field_int(j, UVM_ALL_ON)
    `uvm_field_int(k, UVM_ALL_ON)
  `uvm_object_utils_end

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

  `include "trans_printing.sv"
endclass
```

Example 27 - File: trans1c.sv - no rand outputs - uses field macros - no do_methods()

trans1d - randomized outputs - uses field macros - no ***do_methods()***

The **trans1** class defined in the **trans1d.sv** file has five randomized outputs and five randomized inputs. The **trans1d.sv** example has user-defined field macro definitions but no ***do_methods()***.

```
class trans1 extends uvm_sequence_item;
  rand bit [7:0] a, b, c, d, e; // outputs
  rand bit [2:0] g, h, i, j, k; //  inputs

  `uvm_object_utils_begin(trans1)
    `uvm_field_int(a, UVM_ALL_ON)
    `uvm_field_int(b, UVM_ALL_ON)
    `uvm_field_int(c, UVM_ALL_ON)
    `uvm_field_int(d, UVM_ALL_ON)
    `uvm_field_int(e, UVM_ALL_ON)
    `uvm_field_int(g, UVM_ALL_ON)
    `uvm_field_int(h, UVM_ALL_ON)
    `uvm_field_int(i, UVM_ALL_ON)
    `uvm_field_int(j, UVM_ALL_ON)
    `uvm_field_int(k, UVM_ALL_ON)
  `uvm_object_utils_end

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

  `include "trans_printing.sv"
endclass
```

Example 28- File: trans1d.sv - rand outputs - uses field macros - no do_methods()

trans1e - no randomized outputs - **do_*methods()*** but no calls to **super.do_methods*()***

The **trans1** class defined in the **trans1e.sv** file has five non-randomized outputs and five randomized inputs. The **trans1e.sv** example has user-defined **do_copy()** and **do_compare()** methods but they do not call **super.do_copy()** or **super.do_compare()** respectively. There are no field macro definitions used in this example.

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

       bit [7:0] a, b, c, d, e; // outputs
  rand bit [2:0] g, h, i, j, k; //  inputs

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

  function void do_copy(uvm_object rhs);
    trans1 tr;
    if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_copy() cast")
    {a, b, c, d, e} = {tr.a, tr.b, tr.c, tr.d, tr.e};
    {g, h, i, j, k} = {tr.g, tr.h, tr.i, tr.j, tr.k};
  endfunction

  function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    trans1 tr;
    bit    eq;
    if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_compare() cast")
    eq  = (a == tr.a);  // Compare outputs
    eq &= (b == tr.b);
    eq &= (c == tr.c);
    eq &= (d == tr.d);
    eq &= (e == tr.e);
    eq &= (g == tr.g);
    eq &= (h == tr.h);
    eq &= (i == tr.i);
    eq &= (j == tr.j);
    eq &= (k == tr.k);
    return(eq);
  endfunction

  `include "trans_printing.sv"
endclass
```

Example 29 - File: trans1e.sv - no rand outputs - uses do_methods() - no super.do_methods()

16.4. trans1f - randomized outputs - uses field macros - no UVM_ALL_ON flags

The **trans1** class defined in the **trans1f.sv** file has five non-randomized outputs and five randomized inputs. The **trans1f.sv** example has user-defined field macro definitions but omits the **UVM_ALL_ON** flags and replaces them with **UVM_NOPACK** or **UVM_NOCOMPARE**, which automatically turn on the **UVM_ALL_ON** settings. There are no ***do_methods()*** in this example.

```
class trans1 extends uvm_sequence_item;
       bit [7:0] a, b, c, d, e; // outputs
  rand bit [2:0] g, h, i, j, k; //  inputs

  `uvm_object_utils_begin(trans1)
    `uvm_field_int(a, UVM_NOPACK)     // Same as UMV_ALL_ON | UVM_NOPACK
    `uvm_field_int(b, UVM_NOPACK)     // Turns on UVM_COPY & UVM_COMPARE
    `uvm_field_int(c, UVM_NOPACK)
    `uvm_field_int(d, UVM_NOPACK)
    `uvm_field_int(e, UVM_NOPACK)
    `uvm_field_int(g, UVM_NOCOMPARE) // Same as UVM_ALL_ON | UVM_NOCOMPARE
    `uvm_field_int(h, UVM_NOCOMPARE) // UVM_COPY does not work
    `uvm_field_int(i, UVM_NOCOMPARE)
    `uvm_field_int(j, UVM_NOCOMPARE)
    `uvm_field_int(k, UVM_NOCOMPARE)
  `uvm_object_utils_end

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

  `include "trans_printing.sv"
endclass
```

Example 30 - File: trans1f.sv - no rand outputs - uses field macros - no UVM_ALL_ON flags