UVM - Stop Hitting Your Brother Coding Guidelines

Rich Edelman and Chris Spear
Mentor Graphics, Inc.
8005 Boeckman Rd.
Wilsonville, OR 97070

***Abstract* - This paper show multiple ways to write UVM code, and explains the tradeoffs.**

I. INTRODUCTION

UVM promised a perfect world where a common set of guidelines for testbenches and connected verification IP would make a compatible, simpler world. Just don't look too closely at the standard. UVM carries baggage from previous standards such as OVM, VMM, *e*RM, AVM, and more. This paper describes situations where UVM provides multiple ways to solve a problem, and approaches outside of UVM, explains the issues with certain approaches, and recommends solutions, and describes the tradeoffs.

The authors follow the adage on giving guidance: it is better to tell someone what to do, than what not to do. If you tell a kid to stop hitting his brother, he might not know what to do instead. So first, always tell someone what to do, such as go outside and play. You can then tell what they already know.

BACKGROUND

This paper provides style and coding recommendations. As with all such recommendations, there is no single right answer for everyone. Style is in the eye of the beholder. Coding recommendations usage depends on the willingness to suffer a certain pain level. For many years, a software coding recommendation was "No Gotos". This recommendation relieves a certain complexity and difficulty in understanding and debugging the software. Code with gotos was hard to read and hard to debug and often called "spaghetti code". This recommendation is now baked into programming languages as most no longer include goto statements. Hopefully some of these recommendations in this paper will become part of future versions of UVM.

It is with this in mind and with this same spirit that the style and coding recommendations are listed below.

PROJECT LIMITATIONS

What is the biggest impact on a project schedule? It is not how fast the design code can be written, or the speed of the simulator. The verification code is larger than the RTL, so creating and debugging the testbench and the tests is what is keeping your design from getting to market.

You should write your code as if someone is looking over your shoulder, asking what it does. That person could be another engineer, your manager, or even you in two months when you have to go back and debug a problem with an existing test, or reusing a component on a new environment. Writing clear, well documented code may take more time today, but will save you countless hours later in this project and the next.

If you want to boost the performance of your simulation, run the profiling tools to find the real bottleneck, which is rarely where you expect. Much of your testbench code, such as construction and configuration, only executes once or a few times over an entire simulation, so focus on clarity over performance. Clever code is not always reusable code.

TYPES OF RECOMMENDATIONS

There are different kinds of recommendations, including: "Write code THIS way", "Don't use this class", "Don't write this kind of code", and "Do use this class". The UVM has many different classes and functions and provides a very powerful way to improve verification productivity. But as with all powerful tools, care must be taken. A chainsaw is a powerful tool, and it is always used with a mind for safety and with proper care. The UVM should be approached in the same way: with certain characteristics in mind, including:

- Clarity
- Simplicity
- Ease-of-modification
- Performance

The question of performance recommendations is left as an exercise for the reader or some future paper. Performance is important – but most testbenches first suffer from being hard to read, hard to maintain and hard to debug. Occasionally they can get slow. Once again – if your simulation is too slow, run the profiler, don't sacrifice clarity and reusability.

A. Solid Recommendations

Many of these coding style recommendations are widely known, yet badly written code is still seen in UVM testbenches. You want to reuse code to leverage the efforts of others, but reusing poor code is just asking to repeat past mistakes.

1. Group configuration values into "config objects", which are classes extended from uvm_object. Don't write individual values in the uvm_config_db.
   - Why: Set and get of individual values is error prone, difficult to debug, and hurts performance. The uvm_config_db is a good way to pass virtual interface handles from the RTL code's static domain to the testbench dynamic domain. However, it is poorly suited for passing individual configuration values down through the testbench. The configuration database organizes values based on strings with wildcards. As you store more and more entries, the overhead of this string matching becomes unacceptable. As an example, while a design house was integrating several testbench blocks, the build phase shot up to 24 hours of CPU time, even though the run phase took less than an hour. Profiling revealed the uvm_config_db caused the slowdown.
   - Example: Your TX agent needs an active/passive flag, a base address, a flag to enable the coverage collector, a handle to a virtual interface, and a sequencer handle. Here is a simple agent config object class.

```
// TX agent config object
class tx_config extends uvm_object;
  `uvm_object_utils(tx_config)
  function new(input string name="tx_config");
    super.new(name);
  endfunction

  rand uvm_active_passive_enum active; // Active flag
  rand bit [31:0] base_address;        // Base address
  rand bit        enable_coverage;     // Protocol coverage
  virtual tx_ifc  vif;                 // Virtual interface
  uvm_sequencer #(tx_item) sqr;        // Agent's sequencer
endclass
```

   - Analogy: When you head out to the grocery store to purchase a dozen items, you buy and bring them all home in one trip. You wouldn't make a separate trip between the store and your home for each individual item. Likewise, bag up all the configuration information and pass around just the single handle.
   - Benefit: Switching from writing all configuration variables into the UVM config database to just a few config object handles dropped the design house's build phase CPU time to under a minute. An uvm_config_db with hundreds or thousands of entries is fast, but one with hundreds of thousands of entries is unacceptably slow, especially if the entries have wildcards. Additionally, reading a value from a config object is hundreds of times faster than getting the value from the database.
   - Benefit: When one of the authors (Chris) first learned UVM and passed every configuration variable down into the lower components, he occasionally specified the wrong path, and wasted too much time debugging his testbench, instead of the design. Once he switched to config

objects, he followed a simple pattern of Test->Environment, and Environment->Agent and never had a DB problem.

- Benefit: The agent configuration is a great way to organize all the variables that control the agent's behavior. Want to know how to set an agent as active or passive? Look here. Does the agent have a coverage collector? There should be a variable to enable it in this object.
- Random: Make the configuration properties **`rand`** so you can randomize them. If you don't add this modifier, you won't be able to randomize them in the future.
- Caveat: Some VIP may look for individual values in the database. If this makes the VIP difficult to parameterize from the testbench, you could make an agent-level pseudo-config object with all the published values, and then have environment code extract individual values and set them into the VIP scope. Better yet, ask the VIP provider to improve their code.
- Organization: The topology of your UVM testbench should closely match the RTL code. If your design has 2 TX ports and an RX connection, the UVM testbench should have 2 TX agents and an RX agent. Partition the testbench configuration so there are 2 TX config objects and one for RX. Make an environment config object with environment level variable, such as a flag to enable the scoreboard, and handles to the agent config objects. Better yet, use dynamic arrays of handles for a more flexible environment.

```
// Environment config object
class tx_env_config extends uvm_object;
  `uvm_object_utils(env_config)
  function new(input string name="tx_env_config");
    super.new(name);
  endfunction

  rand tx_config tx_cfg[];     // TX config objects
  rand rx_config rx_cfg[];     // RX config objects
  rand bit enable_scoreboard; // Env-level scoreboard
endclass
```

- Exception: There may be global information that is not specific to an environment or agent, such as the register model handle. Create a global config object to hold these values, and add a reference to it in the environment config object.
- Programming tip: Avoid copying values from the config object into local variables. Whenever you store the same data in multiple places, you are more vulnerable to bugs. You have enough of a challenge debugging hardware issues, so why expose yourself to potential testbench issues too?

2. When uvm_config_db::get() fails to find a virtual interface or config object handle, you should stop simulation with a uvm_fatal message, not a lower severity.
    - Why: The test class gets the virtual interfaces from the uvm_config_db. Each component gets its config object including the virtual interface, from the uvm_config_db. If these are not found, this is a testbench bug and simulation cannot continue. Don't use just the uvm_error macro as the simulation will continue and your code will fail when it uses a null handle. Now you are one more step removed from the original problem.
    - How: Here is code from a base test class to get the virtual interface from the uvm_config_db.

```
if (!uvm_config_db#(virtual tx_ifc)::get(this, "", "tx_if", tx_if))
  `uvm_fatal("FAIL", "Unable to find tx_if in uvm_config_db")
```

    - How: Here is environment or agent code to get the config object from the uvm_config_db.

```
if (!uvm_config_db#(tx_config)::get(this, "", "tx_cfg", tx_cfg))
  `uvm_fatal("FAIL", "Unable to find tx_cfg in uvm_config_db")
```

3. In uvm_config_db::set() calls, only put wildcards on the end of instance names.
   - Why: The uvm_config_db performs string matching to find an entry. If you use wildcards, you increase the number of unintended potential matches. The closer the wildcard character is to the front of the string, the greater number of matches. The worst case is an instance name of just "*", which the above design house used extensively as show here.

```
// Bad UVM code!
uvm_config_db#(int)::set(null, "*", "memory_size", memory_size);
```

   That is equivalent to the following Linux command which can take hours to search the file system. (You know you've done this, but just won't admit it!)

```
% find / -name gcc -print
```

   - Guideline: Wildcard instance names are handy when passing information across multiple lower levels. For example, the environment passes the agent config object, which contains the virtual interface, into the agent and its subcomponents, the monitor and driver. The environment makes the following call to pass the tx_agt_cfg handle. The wildcard name, "agt*" means that the handle is visible to the agent and all scopes below.

```
uvm_config_db#(tx_agt_config)::set(this, "agt*", // Wildcard
          "tx_agt_cfg", tx_agt_cfg);
```

4. Pass config objects inside your testbench with OOP-style set_config() methods, instead of the confusing uvm_config_db. (This replaces the above uvm_config_db recommendations, except passing virtual interface handles from the RTL to the test.)
   - Why: Once you convert your testbench from passing individual values to passing config objects, you can see the bigger picture, which is that a testbench is configured and built from the top down, guided by the configuration objects. These object are created at higher levels and their handle is passed to lower levels. For example, the test passes the environment config object into the environment. This is a simple pattern so why burden yourself with the complexity of the uvm_config_db? Since the test already has a handle to the environment, just pass the handle directly with an OOP-style set() method.
   - Example Here is the code for the base test class and environment class showing how to pass the environment config object. The environment and agent config classes are show above.

```
class tx_test_base extends uvm_test;
  tx_environment env;
  tx_env_config  env_cfg;
  tx_agt_config  tx_cfg;

  virtual function void build_phase(uvm_phase phase);
    env = tx_environment::type_id::create("env", this);
    env_cfg = tx_env_config::type_id::create("env_cfg", this);
    env.set_config(env_cfg); // Pass config to env
    // Rest of build_phase actions
  endfunction

  // Other base test code...
endclass
```

```
class tx_environment extends uvm_environment;
  tx_env_config env_cfg;
  // Receive configuration from test
  virtual function void set_config(input tx_env_config env_cfg);
    this.env_cfg = env_cfg;
  endfunction

  // More environment code...
endclass
```

- Additional: The code to pass agent config object handles from the environment to the agent is similar, along with passing from the agent into the driver and monitor classes.
- Benefits: The call to *handle.set(handle)* is a pattern instantly recognizable to any programmer and is understandable even if you are new to object oriented programming. In contrast, the call to uvm_config_db set() involves a specialization of a parameterized class, calling a static method, and requires that you correctly type five arguments: the type, context handle, instance name, field name, and value. If you get even one of these wrong with the uvm_config_db's set() call, your setting will end up in the wrong place. Worse yet, a mistake in the get() call silently succeeds or fails, with no hint as to what went wrong. (Chris saw an engineer struggle for an hour, debugging a testbench, all because he misspelled the field name. Engineers and misspellings go together like projects and missed deadlines.)
- Caveat: Make sure the environment object is created before you call set_config(). This is unlike uvm_config_db style where you can set the configuration even when no object has been created.
- Caveat: The above code may not be backwards compatible with your existing testbenches. You can easily fix this by having the environment's build phase check the config object's handle and if it is null, call the classic uvm_config_db::get().

```
virtual function tx_environment::build_phase(uvm_phase phase);
  if (env_cfg == null) begin
    // Handle not already set, so look in the DB
    if (! uvm_config_db#(tx_env_config)::get(this, "",
                              "tx_env_cfg", tx_env_cfg))
    `uvm_fatal("NO_CFG", "No config object found")
  ...
endfunction
```

- Caveat: The agent component must pass the config object handle to the monitor and driver components. These can no longer rely on the environment's usage of the wildcard scope in the uvm_config_db set call.
- Caveat: Some sequence code calls uvm_config_dg::get() with a handle to the sequencer, looking for configuration values passed down from the environment or test. You can either have your agent explicitly call uvm_config_db::set() with the sequencer instance name, or use the init_start() as described in item 11.
- Caveat: Technically, this set() style leads to a small increase in the amount of code as you need to write a unique set_config() method for every component. However, the body of method follows a very simple pattern of this.handle = handle, so you are unlikely to make a mistake.

5. Minimize the use of UVM objections and calls to raise_objection() and drop_objection().
   - Why: The primary purpose of the UVM objection is to keep the task-based phases executing, such as run_phase(). Without an objection, UVM ends the phase at the end of the current timeslot. However, if you excessively raise and drop objections, you can cause performance problems. Remember, a single, well planned objection works as well as dozen scattered ones.

- Recommendation: A test must raise an objection before starting a sequence to prevent the run phase from ending. A scoreboard might raise an objection while waiting for the last transactions. But don't raise and drop objections inside a sequence as the test-level one is already doing its job. That one test-level objection is enough to keep the phase running for the entire top-level virtual sequence, all its child sequences, and transactions.

```
virtual task tx_test::run_phase(uvm_phase phase);
  tx_virtual_sequence vseq;
  vseq = tx_virtual_sequence::type_id::create("vseq");
  phase.raise_objection(this);
  vseq.start(null);
  phase.drop_objection(this);
endtask
```

- Reality: The purpose of UVM objections is to keep your test running long enough to verify that the design processed the intended stimulus, and no longer. It is difficult for you to know how long a simulation should run. Is your test done when the last sequence item is sent by the driver into the design? No, as the transaction still needs to propagate through the design. So if you are raising and dropping objections at the sequence level, you still need to extend the phase long enough for that last output transaction to arrive. You can call set_drain_time() to extend the phase for a fixed amount of time, or use the phase call-back method phase_ready_to_end().
- Alternatives: Some companies recommend that you encase every individual sequence item with objections, raising the objection before the start_item() call, and dropping after the finish_item() or get_response() call. This is unnecessary for a non-pipelined protocol as your sequence starts when the first item is started and ends when the last item is finished. If you are verifying a pipeline protocol, consider the set_drain_time() or phase_ready_to_end() methods described above.
- Recommendation: A common testbench bug is when it raises an objection but never drops it. The less you raise objections, the less chance of encountering this bug. Inoculate your testbench by specifying who is raising and dropping the objections. Pass the location of the objection call with the get_full_name() method, as follows.

```
virtual task run_phase(uvm_phase phase);
  ...
  phase.raise_objection(this, get_full_name() );
  vseq.start(null);
  phase.drop_objection(this,  get_full_name() );
  ...
```

6. When starting a sequence item, call the create(), start_item(), randomize(), and finish_item() methods instead of the `uvm_do* macros, the "training wheels" of UVM.
   - Why: The macros are great for beginners, helping you write a simple sequence in just a few lines. But as soon as you want to do something more complex, like changing the randomization results, you will have to learn the individual steps. If you try to manually expand the macros, you can be overwhelmed as the macros are heavily layered, difficult to reverse engineer, and call obscure methods. The macros were designed to be easy for new users but run out of steam when you want to try something different.
   - Instead: Once you understand the base methods, you can build sequences that perform complex actions. Learn the four steps and you will be creating complex stimulus with ease.
   - Benefits: Your sequence code will be easy to write and maintain. Say you wanted to create 10 sequence items and randomize them with dependencies between the objects, such as each having an address that is larger than the previous one. This is easy with a few foreach-loops traversing an array of handles, but impossible to write with the uvm_do macros.

7. Write your sequence item classes quickly with the UVM field macros instead of the sequence item do_*() methods.
    - Why: More code means more bugs.
    - Benefit: The field macros allow you to automatically create hundreds of lines of code with a single macro, such as uvm_field_int. Even a simple set of do_*() methods plus the convert2string() method requires dozens of lines of code. You can quickly create a base sequence item class that easily fits on one page and is easy to understand.
    - Benefit: The default sprint() method is created automatically. Writing a convert2string() by hand can take an hour or three if you want to precisely line up every field. That is time taken away from creating new tests and sequences.
    - Caveat: If your IP already has the do_*() methods, stick with them.
    - Caveat: For complex transactions, see the next recommendation.
8. Write your sequence item classes accurately with the sequence item do_*() methods instead of the field macros.
    - Why? The do_*() methods allow you to precisely control how your transaction fields are manipulated. If your sequence item class has properties that are conditional on other fields, such as a type, you will only be able to copy and compare them by explicitly writing the do_copy() and do_compare() methods. The field macros can't handle this case.
    - Benefit: You can create a convert2string() that precisely formats the transactions properties for maximum readability. This can reduce debug time, compared to the tabular print style from the field macros.
    - Caveat: If your IP already has the field macros, stick with them.
    - Performance: While it is possible to measure a slight performance advantage of do_*() methods compared to the field macros, this difference is noise in a real testbench running with RTL. If your simulations run too slow, use the profiler to find the real bottleneck, instead of guessing in the dark.
9. When you need to access properties in the UVM base classes, call the provided set and get methods, instead of accessing them directly. The classic example is that many sequences reference m_sequencer in order to access the uvm_config_db through a component handle. Instead, call the get_sequencer() method which has been in UVM from the beginning.
    - Why: If your code calls these set and get methods, it stays independent of any specific implementation of UVM. The 1800.2 standard describes the behavior of the base class library, but each EDA vendor will create their own implementation the library. Each will have its own members.
    - Example: If your sequence needs the handle to its associated sequencer, call get_sequencer(), a method that has been in the library from UVM 1.1c in 2011. The handle m_sequencer in uvm_sequence_item is a protected property, which should tell you that the developers didn’t want you to touch it. And don’t even ask about the p_sequencer macro!
10. Call the factory create() method for anything that you want to override, such as transactions and components.
    - Why: Use the power of the uvm_factory to make your testbench classes more configurable. The factory allows you to inject new behavior into components without having to make any changes to the class.
    - Instead: When you need to construct an object, always call class::type_id::create() or the `uvm_create macro.
    - Exception: Call the SystemVerilog constructor new() for classes that are never extended such as uvm_sequencer, TLM components, uvm_reg_predictor.
    - Additionally: The create method’s name argument becomes the instance name of the component. Make debugging easier by keeping the name and the handle the same.
    - Improvement: The following macros create UVM objects and components, and are shorter than the equivalent handle = class::type_id::create(“name”) style. If the UVM standard was updated to add a dummy second argument to uvm_object create, only one macro would be needed.

```
`define my_create_o(_type, _name) \
   _name = _type::type_id::create(`"_name`")
`define my_create_c(_type, _name, _parent=this) \
   _name = _type::type_id::create(`"_name`", _parent)

tx_item t;
tx_agent agt;
`my_create_o(tx_item, t);    // t = tx_item::type_id::create("t")
`my_create_c(tx_agent, agt); // agt =
                             //tx_agent::type_id::create("agt",this)
```

11. Make your sequences easier to run with a user-defined virtual method. Define your own arguments including responses.
    - Why: A configurable sequence needs parameters such as the sequencer handle, number of transactions, randomization constraint weights. Pass these directly to a method in the sequence.

```
class tx_seq extends uvm_sequence #(tx_item);
  int num_xacts;
  virtual task init_start(input uvm_sequencer #(tx_item) sqr,
                          input int num_xacts,
                          output uvm_status_e aok);
    this.num_xacts = num_exacts; // Parameter to seq
    this.start(sqr);             // Start this sequence
    aok = UVM_IS_OK;             // All is cool
  endtask
endclass
```

    - Benefit: You can make the sequence easier to run by wrapping these details in the method. Now even the software people on the team can create tests and virtual sequences by calling these methods, without having to learn UVM details.
    - Example: The UVM Register Layer's methods such as read() and write() are great examples of high-level methods that start sequences under the hood.
12. If your test runs more than one sequence, put them into a virtual sequence.
    - Why: This novel combination is now reusable. A test is built from smaller steps to create, apply, and check stimulus.
    - Benefit: If you write a test that applies several flavors of sequences by manually starting each base sequence, that combination cannot be reused. If you instead combine them into a virtual sequence, your test and other can reuse it. Better yet, it can be reused in higher level virtual sequences.
13. Check SystemVerilog operations
    - Checking: An often overlooked fatal simulation problem is SystemVerilog randomization when there is a constraint conflict. If you try to randomize an object and it fails, by default the simulator does not print any message, and the random variables retain their previous values. So you might send the same transaction values over and over without realizing it. (Some simulators have switches to stop simulation and create test cases – always run with these!)
    - Guideline: Check the result from the randomize function with an if-statement and write a fatal error with a meaningful message.

```
if (!tx_obj.randomize())
  `uvm_fatal("FAIL", "tx_obj.randomize()")
```

    - Warning: Don't check the status with a SystemVerilog assertion. If you disable assertions, the contents of the assertion are not evaluated, which means your code won't call randomize(). This causes a “silent-but-deadly” bug where no error or warning messages are printed, yet the stimulus is incorrect. Why would you ever disable assertions? Some engineer always tries this as a way to improve simulation performance.

- Corollary: Also check every SystemVerilog $cast() call, just in case, and end simulation with a fatal error. If you don’t check the return value, and the $cast function fails, you will get an error from the simulator that might be hard to understand. Also, when you make the effort to write a meaningful message, you will take a second look at the code, and might spot a problem.

REFERENCES

[1] “Slaying the UVM Reuse Dragon. Issues and Strategies for Achieving UVM Reuse,” by M. Baird, R Oden, DVCON 2016.
[2] “Using UVM Virtual Sequencers & Virtual Sequences,” C. Cummings, J. Bergeron, DVCON 2016.
[3] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, IEEE, New York, NY, IEEE Std 1800™-2017.
[4] “Verification Academy UVM Cookbook” https://verificationacademy.com/cookbook.
[5] “Universal Verification Methodology (UVM) 1.2 Class Reference”, Accellera, June 2014.

APPENDIX I – A SIMPLE MONITOR

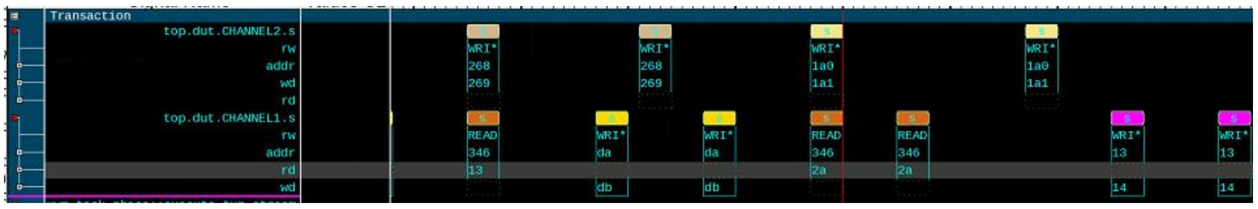
The simple monitor code below demonstrates some of the recommendations above, including using simple macros, using the normal UVM phases and convert2string(). Additionally, the monitor is written in a reusable way – connected on one side to a virtual interface (vif), and connected on the other side to an analysis port, publishing any transaction monitored for other verification components to process (such as coverage collectors or checkers).

```
typedef enum bit[1:0] {READ, WRITE, IDLE} rw_t;

class tx_monitor extends uvm_monitor;
  `uvm_component_utils(tx_monitor)

  virtual my_if vif;
  uvm_analysis_port #(tx_item) ap;

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

  virtual function void build_phase(uvm_phase phase);
    ap = new("ap", this);
  endfunction

  virtual task run_phase(uvm_phase phase);
    forever begin
      @(posedge vif.CLK);
      if ((vif.VALID == 1) && (vif.READY == 1)) begin
        t = tx_item::type_id::create("t");
        t.rw = vif.rw;
        t.addr = vif.addr;
        if (vif.rw == READ) begin
          @(negedge vif.READY);
          t.data = vif.rd;
        end
        else if (vif.rw == WRITE) begin
          t.data = vif.wd;
        end
        `uvm_info(get_type_name(), $sformatf("Got %s",
          t.convert2string()), UVM_MEDIUM)
        ap.write(t);
      end
    end
  endtask
endclass
```

Appendix II – A Simple Transaction

The transaction definition below demonstrates some of the recommendations above, including simple macro usage with simple basic constraints and convert2string(). Additionally, do_record() is defined and uses a simple UVM macro which allows for faster type based transaction recording (rather than slower, string based recording). The image below the code snippet contains two streams with transactions on them.

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

  rand rw_t        rw;
  rand bit  [31:0] addr;
  bit       [31:0] data;

  rand int duration;

  constraint duration_length {
    duration > 3; duration < 10;
  };

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

  virtual function string convert2string();
    return $sformatf("[%s] rw=%s, addr=%0d, data=%0d (%0d)",
      get_type_name(), rw.name(), addr, data, duration);
  endfunction

  virtual function void do_record(uvm_recorder recorder);
    super.do_record(recorder);
    `uvm_record_field("name", get_name())
    `uvm_record_field("rw", rw.name())
    `uvm_record_field("addr", addr)
    `uvm_record_field("data", data)
    `uvm_record_field("duration", duration)
  endfunction
endclass
```


Transaction
top.dut.CHANNEL2.s
rw
addr
wd
rd
top.dut.CHANNEL1.s
rw
addr
rd
wd
WRI*
268
269
1a0
1a1
READ
346
13
2a
da
db
14