SystemVerilog Interface Classes – More Useful Than You Thought

Stan Sokorac
stan.sokorac@arm.com
ARM Inc., 5707 Southwest Pkwy, Building 1 Suite 100, Austin, TX 78735

***Abstract*- Interface classes, not to be confused with similarly named 'interfaces', were introduced in SystemVerilog 2012, but have seen little adoption in the verification community. While this construct is well established in the software development world, most verification engineers either do not know about it or do not see any benefit in using it. This paper attempts to demonstrate the value of interface classes by sharing some of the most important uses in the verification of the next-generation ARM® Cortex-A® CPU core.**

Keywords- SystemVerilog, Interface Classes

I. Introduction

The concept of an "interface", which is what interface classes are commonly known as in the programming world, was popularized by Java, a widely used software development language [1]. However, the original idea was introduced by the creators of Objective-C, which defined a concept of a "protocol", which inspired the Java interface [2]. Today, most strongly-typed languages support the interface concept – some more recent examples include C# [3], Swift [4], and D [5]. All of these languages feature standard libraries that make heavy use of interfaces, making them a well-understood and commonly used feature.

SystemVerilog introduced the interfaces classes in its 2012 version [6]. However, the popular standard libraries at the time (OVM, VMM, and UVM) were already developed using SystemVerilog features from previous years, and worked around the lack of interfaces through macros and other design patterns. This severely restricted the adoption of interface classes, making them an underused and under-appreciated feature even three years after their inclusion in the spec.

During the development of a new testbench for the next generation ARM® Cortex-A® CPU core, we have relied on interface classes in many different ways, improving the flexibility and quality of the testbench, while furthering its maintainability and debuggability. The purpose of this paper is to present some of these use cases, in an effort to inspire verification engineers to use them in their future efforts, leading to wider adoption and development of new ideas.

II. Use Cases for Interface Classes

A. Observer (Subscriber) pattern

The concept of a *listener* (also commonly known as *observer*), has been used in testbenches ever since the first physical interface monitor was built, even if they were not called by that name. The concept is straightforward – a monitor sees something on the interface, puts it together into some kind of a transaction data structure, and sends it out to interested parties (*listeners*), such as scoreboards and checkers.

A common limitation in early testbenches was the ability to easily publish this transaction to multiple observers. If a testbench had a scoreboard and two checkers that needed this data, the code was written to manually deliver that data to each one of those testbench components, as seen in Fig. 1.

```
function void notify_observers(resolve_txn resolve);
  m_scoreboard.notify(resolve);
  m_checker1.notify(resolve);
  m_checker2.notify(resolve);
endfunction
```

Figure 1. Custom-written code for notifying observers

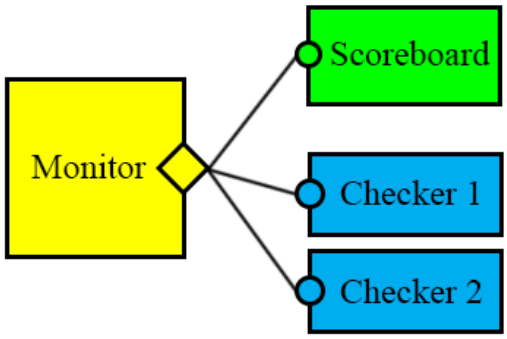

Figure 2. UVM analysis port provides a flexible one-to-many connection between components

UVM has further developed this pattern by creating TLM analysis ports, giving us a way to create generic connections between publishers and subscribers. In Fig. 2, a monitor creates an analysis port, and each subscriber creates an analysis export (*write()*) function, and we have a seamless one-to-many connection [7].

While this setup provides a valuable abstraction when dealing with connections between major components developed by separate teams, or language boundaries, it also has many limitations when used in a smaller SystemVerilog-only testbench. First, the connections are static – they are determined during the environment setup, and only UVM components can partake in the analysis port communications. Second, the communication is limited to a transfer of a single transaction of one (base) type. And, finally, a component that subscribes to multiple analysis ports has to resort to either UVM macro contortions, or create a hierarchy of children to listen to transactions.

The implementation of the subscriber pattern using interface classes solves all three of these problems. Let us start by describing what such an implementation looks like. Fig. 3 shows an example of something that provides very similar functionality to a UVM analysis port. In Appendix A, this example is broken down line-by-line for an in-depth explanation.

```
interface class resolve_listener;
  pure virtual function void new_resolve(arm_txn_resolve resolve);
endclass

class monitor extends uvm_component;
  local resolve_listener m_resolve_listeners[$];

  function void add_listener(resolve_listener listener);
    m_resolve_listener.push_back(listener);
  endfunction

  virtual task run_phase(uvm_phase phase);
    forever begin
      arm_txn_resolve resolve = get_next_resolve();
      foreach(m_resolve_listeners[i])
         m_resolve_listeners[i].new_resolve(resolve);
    end
  endtask
endclass

class resolve_checker extends uvm_component implements resolve_listener;
  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    m_config.monitor.add_listener(this);
  endfunction

  virtual function void new_resolve(arm_txn_resolve resolve);
    if (resolve.is_abort())
      `arm_fatal("Aborts are not expected")
  endfunction
endclass
```

Figure 3. Sample code of functionality similar to UVM's analysis port, using interface classes

The first thing to notice is that this implementation uses *dynamic* connections. The subscriber can register itself with the producer at any point in the simulation and start listening to transactions. We have chosen to do it in the connect phase in the above example, because that is still the best location for static components. However, where the dynamic property of connections becomes very powerful is in stimulus generation. By allowing UVM sequences to directly listen to monitors, BFMs, and scoreboards, reactive sequences are much easier to write, maintain, and understand. Contrast this with a traditional UVM implementation that requires the sequencer (a component) to listen to every analysis port out there that any sequence might need, and then create communication channels between sequencer and sequences to feed that information as needed.

```
task run_sequence();
  m_done = 0;
  m_config.monitor_l1l2.add_listener(this);
  wait(m_done);
  m_config.monitor_l1l2.remove_listener(this);
endtask

virtual function void new_l1l2_request(arm_txn_l1l2 req);
  // Wait until a request to upgrade line from shared to exclusive is seen and
  // send a snoop request to steal the line away
  if (!m_done && l1l2.req_type() == READ_UNIQUE_HIT_SHARED) begin
    send_snoop(SNOOP_INVALIDATE, l1l2.req_address());
    m_done = 1;
  end
endfunction
```

Figure 4. An example of a reactive sequence using subscriber pattern to listen to a monitor directly.

The second thing to notice is that the interface between the listener and the publisher is not limited to a single transaction transfer. The interface function *new_resolve(…)* can send back any additional information that could be useful to a listener without having to wrap it up into another transaction class.

A common use in our testbench is for the BFM to provide a reference pointer to the micro-op that the message relates to. Now, the function changes to the declaration in Fig. 5, which is now a much richer message. The listener still has the original *resolve* transaction, along with context information that it can use to interpret and react to the message, as in code shown in Fig. 6.

The interface can be enhanced even further. We are not limited to only one kind of a message. For example, the micro-op BFM keeps track of many types of events in the lifetime of a micro-op, and can transmit these state changes through several function calls, as shown in Fig. 7.

Finally, implementing a listener that subscribes to multiple producers comes naturally due to the fact that implementation of multiple interfaces is natively supported by SystemVerilog. An example shown in Fig. 8 is a checker that listens to micro-ops being dispatched, as well as requests made to L2 cache, and ensures that they are made in the right order.

```
pure virtual function void new_resolve(arm_txn_uop uop, arm_txn_resolve resolve);
```

Figure 5. A notification function that transfers more than just the transaction.

```
class resolve_checker implements resolve_listener;
  virtual function void new_resolve(arm_txn_uop uop, arm_txn_resolve resolve);
    if (uop.is_load() && resolve.is_clean())
      check_load_data(uop);
  endfunction
  …
endclass
```

Figure 6. A sample listener that makes use of multiple objects being sent through the notification function.

```
interface class uop_listener;
  pure virtual function void new_resolve(arm_txn_uop uop, arm_txn_resolve resolve);
  pure virtual function void new_commit(arm_txn_uop uop, arm_txn_commit commit);
  pure virtual function void new_issue(arm_txn_uop uop);
  pure virtual function void uop_flush(arm_txn_uop uop, flush_cause_e cause);
endclass
```

Figure 7. An notification interface featuring multiple functions for multiple events.

```
class ordering_checker extends arm_checker implements uop_listener, ace_listener;
  local arm_txn_uop m_ordered_uops[$];

  // Register ourselves with micro-op and ACE agents
  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    m_config.uop_agent.add_listener(this);
    m_config.ace_agent.add_listener(this);
  endfunction

  // On commits, record micro-ops that need to be ordered
  virtual function void new_commit(arm_txn_uop uop, arm_txn_commit commit);
    if (commit.is_clean() && uop.is_ordered())
      m_ordered_uops.push_back(uop);
  endfunction


  // On ACE requests, compare address and size
  virtual function void new_ace_req(arm_ace_req ace_request);
    arm_txn_uop uop;
    if (!ace_request.needs_to_be_ordered())
      return;

    uop = m_ordered_uops.pop_front();
    check(ace_request.addr().equals(uop.addr()) && (ace_request.size() == uop.size()),
        {"ACE request seen doesn't match the oldest micro-op: ", uop.covert2string()});

  endfunction
endclass
```

Figure 8. An example of a checker that subscribes to two different monitors.

The resulting code is much cleaner and more straightforward than an implementation using UVM analysis port macros. It is obvious which function does what, and tracing through the code or a stack-trace is as easy as any simple function call between two classes.

In some cases, the listener interface class for complex scoreboards has grown to have a number of function definitions, and not all are relevant to all listeners. One solution is to break up interface classes into smaller ones, but that adds code to those listeners that do need to listen to all events. An elegant solution we used is a "mixin pattern" [8] combined with interface classes. The mixin itself provides empty implementations for all functions inside the interface class, allowing the subclass to only override the ones it needs, as shown in Fig. 9.

A great thing about this pattern is that it can be chained together to still allow us to subscribe to multiple interfaces using mixins for each one. The declaration for our strongly ordered checker can be written as shown in Fig. 10.

```
class uop_listener_mixin(type T = uvm_component) extends T implements uop_listener;
  virtual function void new_resolve(arm_txn_uop uop, arm_txn_resolve resolve);
  endfunction

  virtual function void new_commit(arm_txn_uop uop, arm_txn_commit commit);
  endfunction

  virtual function void new_issue(arm_txn_uop uop);
  endfunction

  virtual function void uop_flush(arm_txn_uop uop, flush_cause_e cause);
  endfunction
endclass

class uop_checker extends uop_listener_mixin#(arm_checker);
  virtual function void new_issue(arm_txn_uop uop);
    check_uop(uop);
  endfunction
endclass
```

Figure 9. An example of using mixin pattern with interface classes.

```
class strongly_ordered_checker extends uop_listener_mixin#(
                                          l1l2_listener_mixin #(arm_checker));
```

Figure 10. A chained declaration of multiple mixins.

B. *Pseudo-Multiple-Inheritance*

The lack of true multiple inheritance can sometimes be limiting. However, interface classes sometimes allow us to work around this limitation. An example of such use in our testbench comes from the micro-op class hierarchy. Fig. 11 shows a fairly typical setup for such a hierarchy – on one side we have addressable micro-ops, such as loads and stores, and on the other, we have some non-addressable ones, like data barriers (*DMB, DSB*). This makes sense, looks great, and everybody is happy until we get to Load-Acquire (*LDAR*) and Store-Release (*STLR*) instructions. For those not familiar with ARM instruction set, *LDAR* and *STLR* behave as sort of a two-in-one instructions – they are both loads/stores, as well as barriers. Where in this hierarchy do they fit in?

If we had multiple inheritance available, LDAR could simply inherit from both Load and DataBarrier classes. Since we do not, most class hierarchies will let LDAR inherit from Loads, and either place all barrier-specific function inside the base micro-op class, or write "special case" code everywhere to deal with this – in both cases, resulting in code that is harder to maintain[1].

However, interface classes allow us to do something similar to multiple inheritance. We can define a *Barrier* interface class that declares the functions that describe the barrier behavior, and then have *DataBarrier*, *LDAR*, and *STLR* classes implement it. Now, a barrier check is a simple *$cast* test, after which we have our well-defined interface to inspect the barrier behavior.

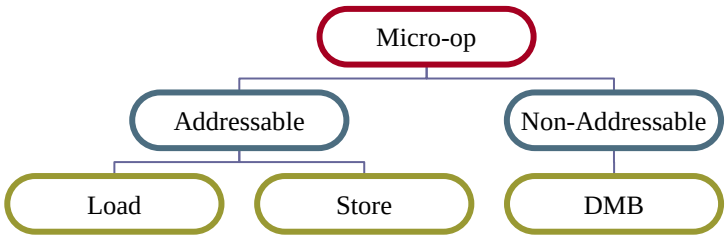

Figure 11. Typical class hierarchy of CPU instructions types

[1] Putting all functions in the base class is undesirable, as the class ends up being polluted with dozens of functions that are not needed there.

```
interface class barrier;
  // Return 1 if this barrier affects the given uop in a given direction
  pure virtual function bit affects_uop(arm_txn_uop uop, dir_e direction);

  // Perform age comparison between a barrier and a uop
  pure virtual function bit is_barrier_older(arm_txn_uop uop);
…
endclass

class barrier_checker;
  function void check_out_of_order_resolve(arm_txn_uop first, arm_txn_uop second);
    barrier bar;
    if ($cast(bar, second) && bar.affects_uop(first, YOUNGER))
      `arm_fatal("Uop bypassed a barrier it isn't allowed to.")
  endfunction
endclass
```

Figure 12. An example of 'barrier' interface class use.

This can then be further developed by implementing interfaces for other micro-op attributes that exist in various parts of the hierarchy, such as, for example, exclusives[2].

C. Data Serialization

For the purpose of debugging or stimulus analysis, our testbench has a mode in which it records every interesting event into an SQL database. With custom tools for analysis and reporting, we have a powerful way of investigating exactly what happened in a test.

Most recorded events are simply transactions seen on various interfaces. However, we also record debug messages, and even checker and scoreboard hints that facilitate better understanding of the relationships between transactions. In order to support a generic way of recording all of these events into a database, we have defined an *arm_event* interface in Fig. 13.

To automatically record all transaction, our base transaction class implements this interface with some default values, and registers itself with the central event recorder. Concrete transaction class implementations then fill-in the details, for which we have created a macro similar to *`uvm_object_utils* to make it very easy to do so, as shown in Fig. 14.

A keen observer would notice that the code in Fig. 14 would work the same way if we simply created an *arm_event* base class that extends *uvm_sequence_item*, without any use of interface classes. However, with *arm_event* being an interface, any unrelated class can register itself, since it can implement the interface without modifying its base class. For example, a debug message class, a member of a different branch of the class hierarchy, can record itself inside the database (Fig. 15).

```
interface class arm_event;
  pure virtual function int     event_id();
  pure virtual function time_t  event_timestamp();
  pure virtual function cpu_t   event_cpu();
  pure virtual function string  event_location();
  pure virtual function string  event_type();
  pure virtual function string  event_subtype();
  pure virtual function string  event_description();
endclass
```

Figure 13. An Event interface class, defining basic fields in the event database.

[2] *LDREX*, *LDAXR*, *STREX*, *STLXR*, etc. are all instructions that take on properties of loads/stores, barrier, *and* exclusives.

```
class arm_txn extends uvm_sequence_item implements arm_event;
  function new(string name="");
    super.new(name);
    arm_event_recorder::get_instr().register_event(this);
  endfunction

  virtual function int     event_id();
    return m_id;
  endfunction

  virtual function time_t event_timestamp();
    return m_timestamp;
  endfunction

  virtual function string event_description();
    return convert2string();
  endfunction
 ...
endclass

class arm_txn_l1l2_req extends arm_txn;
  `arm_event_utils("l1l2_req",    // type
                   req_type(),    // subtype
                   "intf_l1l2")   // location
 ...
endclass
```

Figure 14. A common implementation of the Event interface used by all transactions.

For the majority of the events, this is sufficient to give us all the information we need to debug a test. However, some events require a richer set of fields. Micro-ops are a very good example – in addition to the physical address, most micro-ops also have a virtual address, source and destination registers, source data, and result data. Ideally, we would like to create a new table inside our database to hold this extra information. Interface classes once again provide us with generic way to allow any class to create and populate a new table, an example of which is shown in Fig. 16.

In this example, we create a *dumpable* interface that any event can implement, through which it can provide SQL commands to create and populate the new table. The event recorder then performs a test *$cast* on each event to see whether the *dumpable* interface is implemented, and executes the SQL commands if it is. With this setup, any event can create a new table by simply implementing an interface, without any changes to the generic event recorder.

```
class arm_debug_message extends arm_message implements arm_event;
  `arm_event_utils("debug_msg", tag(), parent().name())
  function new(uvm_component parent, string tag, string description);
    super.new(parent, tag, description);
    arm_event_recorder::get_inst().register_event(this);
  endfunction
  ...
endclass
```

Figure 15. A non-transaction class can implement the event interface.

```
interface class dumpable;
  pure virtual function string sql_create_table();
  pure virtual function string sql_insert();
endclass

class virtual_address_txn implements arm_event, dumpable;
  virtual function string sql_create_table();
    return "CREATE TABLE IF NOT EXISTS virtual_txn(id PRIMARY KEY, va, srca, srcb);"
  endfunction

  virtual function string sql_insert();
    return "INSERT INTO virtual_txn VALUES(m_id, m_va.to_int(), m_srca, m_srcb);"
  endfunction
  ...
endclass

class event_recorder;
  ...
  function void dump_to_db();
    foreach(m_events[i]) begin
      dumpable d;
      if($cast(d, m_events[i]))
          execute_sql({d.sql_create_table(), d.sql_insert});
    end
  endfunction
endclass
```

Figure 16. A flexible way of allowing any class to define and populate a new SQL table.

D. *Object Clocking*

In a typical testbench, many components will require access to the clock, with run phases often featuring a *forever @clk* loops. One of the downsides of such a distributed clocking approach is that test random stability suffers due to the fact that clock events can be observed by components in different order from test to test. One approach to improving the clocking mechanism is have one central forever loop that calls a *clock* (or *tock*) function in every component in the system in a predictable order.

This approach typically requires a new base component class to be defined with the "tock" function, which is often undesirable[3], or having the central clocking mechanism understand what all kinds of classes need to be clocked. The base component class breaks down even further if there are any dynamic objects that require clocking, as now we need a base object class with "tock" function, as well. All of these approaches lead to some clumsy code.

Interface classes clean this up significantly. We set up a *clockable* interface class (Fig. 17), which defines the *tock* function. Now, any component or even object that wants to receive the *tock* event can simply implement the interface and register itself with the central clock distributor which holds a single ordered queue of *clockable* objects.

III. Conclusion

SystemVerilog interface classes are heavily used in current generation CPU verification at ARM®. This paper presents four major use cases that make testbench development easier. The subscriber pattern makes the message passing cleaner and more flexible, which has a particularly positive impact on stimulus quality where sequences adapt to events coming from the design's monitors. The multiple inheritance pattern simplified our class hierarchies. And, finally, data serialization features encourage better off-line tool development and enhancement.

Going forward, I hope to see increased adoption of interface classes in wider verification community, which will inevitably lead to new ideas for their use, and eventual inclusion in standard libraries, such as UVM.

[3] Many UVM classes, such as uvm_monitor, uvm_sequencer, etc. are subclassed from uvm_components. By creating a custom uvm_component base class, the hierarchy diverges into two branches – the original components, and the custom ones. This leads to either custom modifications of the UVM library, or redefinition of many components in the "new" branch.

```
interface class clockable;
  pure virtual function void tock();
endclass

class clocking_center extends uvm_component;
  struct {
    clockable obj;
    string name;
  } clockable_t;
  local clockable_t m_clockables[$];
  function void register_clockable(clockable obj, string name);
    sorted_insert(m_clockables, '{obj, name});
  endfunction

  task run_phase(uvm_phase phase);
    forever @clk
      foreach(m_clockables[i])
        m_clockables[i].obj.tock();
  endtask
endclass

class interface_timeout_monitor extends uvm_component implements clockable;

  local int m_counter = 0; // count the number of idle cycles

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    clockable_center::get_inst().register_clockable(this);
  endfunction

  virtual function void tock();
    if (intf.idle()) begin
      m_counter++;
      if (counter > TIMEOUT)
        `arm_fatal("Interface idle for too long.")
    end else
      m_counter = 0;
  endfunction
```

Figure 17. *Clockable* interface class provides a flexible method for any object to subscribe to the clocking *tock* event.

REFERENCES

[1] "What Is an Interface? The Java™ Tutorials", https://docs.oracle.com/javase/tutorial/java/concepts/interface.html

[2] James Gosling, and Henry McGilton, Sun Microsystem Computer Company, "The Java Language Environment: A White Paper", p. 54.

[3] "interface (C# Reference)", https://msdn.microsoft.com/en-us/library/87d83y5b.aspx

[4] "The Swift Programming Language (Swift 2.1): Protocols", https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html

[5] "D Programming Language: Interfaces", https://dlang.org/spec/interface.html

[6] IEEE Computer Society, "IEEE Standard for SystemVerilog", IEEE 1800-2012, pp. 157-166.

[7] Accellera, "Universal Verification Methodology (UVM) 1.1 Class Reference", June 2011, pp. 231-234.

[8] Tudor Timi, "Fake It 'til You Make It - Emulating Multiple Inheritance in SystemVerilog", http://blog.verificationgentleman.com/2014/09/emulating-multiple-inheritance-in-system-verilog.html

APPENDIX A

In this section, the code from Fig. 3 is broken down line-by-line to explain the mechanics of declaring and using an interface class in more detail. The implementation starts with a declaration of an interface class *resolve_listener*, which defines, in this case, a single *new_resolve* function. This is a 'contract' that says that any class that wants to call itself a *resolve_listener* must implement a *new_resolve* function that looks exactly like the one declared in the interface class:

```
interface class resolve_listener;
  pure virtual function void new_resolve(arm_txn_resolve resolve);
endclass
```

The function is declared as *pure virtual* because interface classes do not provide implementations of functions. That is the job of a class that implements it.

We next define our producer *monitor*, a class that sees the *resolves* happening in the system and communicates the details to all interested parties. The key structure in the *monitor* class is a queue of *resolve_listener* objects, where *monitor* keeps track of who is listening to *resolve* events:

```
  local resolve_listener m_resolve_listeners[$];
```

The *run_phase* illustrates how this queue is used. On each new *resolve* seen by the BFM, it loops through the list of listeners and calls the *new_resolve* function, which is analogous to an analysis port write in UVM:

```
      foreach(m_resolve_listeners[i])
         m_resolve_listeners[i].new_resolve(resolve);
```

The *resolve_checker* class demonstrates what happens on the receiving side of this call. The class is declared as a *uvm_component* that also promises to implement functions from the *resolve_listener* interface class:

```
class resolve_checker extends uvm_component implements resolve_listener;
```

The implementation of the *new_resolve* function in our example illustrates that the *new_resolve* function looks no different from any other virtual function. It takes in the parameter *resolve* sent by the *monitor*, and performs a check on its properties:

```
  virtual function void new_resolve(arm_txn_resolve resolve);
    if (resolve.is_abort())
      `arm_fatal("Aborts are not expected")
  endfunction
```

The final piece is the connection between our consumer, *resolve checker*, and our producer, *monitor*. We chose to make that connection in the *connect_phase*, using *monitor*'s *add_listener* function, although this connection could have been made at any point in the simulation.

```
    m_config.monitor.add_listener(this);
```

Note that we are passing *this* as the listener parameter to the *add_listener* function. Even though *this* is of type *resolve_checker*, with a base class *uvm_component*, *monitor* will accept it as a *resolve_listener*, because the class implements that interface and can therefore also be used as a *resolve_listener* type. This reinforces the flexibility of interface classes – *monitor* does not need to know what kind of an object is registering themselves as a listener. As long as it implements the *resolve_listener* interface, the receiver of the *new_resolve* call could be a checker, a scoreboard, some other component, or even a dynamic object.