Using SystemVerilog Interfaces and Structs for RTL Design

Tom Symons
Hardware Advanced Development
Oracle Labs.
Austin, Texas. U. S. of A.
tom.symons@oracle.com

Nihar Shah
Hardware Advanced Development
Oracle Labs.
Austin, Texas. U. S. of A.
nihar.shah@oracle.com

***Abstract*— System verilog interfaces and structs have many useful benefits in RTL design, but they have not been readily adopted due to limited support by the EDA vendors. We used SystemVerilog interfaces and structs in our recent project, and we have recently taped out a chip with some modest usage of these in RTL. This paper discusses the benefits of SystemVerilog interfaces and structs in RTL, the tools we used and the issues we faced, and how we worked around those issues. Furthermore, we propose improvements to the toolset and standards that will improve the adoption of these beneficial constructs.**

Keywords—SystemVerilog; interfaces; structs; RTL; hardware design; UVM; verification

I. INTRODUCTION

SystemVerilog interfaces have proven to be indispensible for verification. However, their use for RTL design has been limited at best. Structs are equally useful and are less problematic, but still are not commonly used in RTL design. The primary reason interfaces and structs are not used more in RTL design is because there are still some tools that have limited support for them. But support for SystemVerilog constructs in RTL has been steadily increasing, and it is becoming more common to hear of successful tool flows utilizing SystemVerilog design constructs. But you still must verify your flow across your toolset and develop a usage methodology that maps to the tools and common design needs. This paper describes a tool set and methodology that supports using interfaces and structs in RTL design.

II. BENEFITS OF INTERFACES AND STRUCTS IN RTL DESIGN

Interfaces are useful in RTL design for the following reasons:

1. They can dramatically collapse the size of netlist files. A modest SOC can easily have 2500 signals just in one netlist file, translating to roughly 3x2500=7500 lines (one line for wire declaration, one to connect to source module and one to connect to destination module). If the average interface has 10 signal names, then about 250 interfaces are required and need only 3x250=750 lines in the netlist file. That's a saving of 6750 (92%) lines of code. In our SOC design, 25% of all lines were used for just interconnection.
2. Having a group of signals bundled into an interface can also ease debug considerably, particularly for users who have to debug sections of the code they do not deal with everyday. The grouping of signals into an interface eliminates the confusion of which signals belong to which bus and allows the user to drag the entire interface into a wave viewer and see all the signals nicely grouped together.
3. Using the same interface throughout the design is a great way to ensure that all designers use the same signal names for all signals in a given protocol. This also makes it easy to reuse any assertions or functional coverage defined for the bus.
4. Adding/removing/renaming a signal in an interface need only be done in the interface and the two endpoints. There is no need to update all the connections up and down the module hierarchy between the two endpoints. This can save some tedious, error prone work when signals traverse many module boundaries, as well as eliminating a potentially huge number of lines of netlisting code across all those modules.
5. Interfaces are a great place for assertions that check for the validity the interface signals. This ensures that the assertions are always used on each instance of the interface and helps avoid duplication of those assertions.
6. Interface provide an encapsulation for user-defined debug aids. Additional signals can be added to an interface such as path name or transaction count which can be viewed in a waveform to enhance debug.

Writing, reading and debugging signals may seem like a trivial task. But as chips get larger and larger, the number of signals to manage escalates dramatically. Managing each signal explicitly just does not scale well into these large designs. The advantages of interfaces may not seem apparent when just a few interfaces are used, but when a large number of interfaces are used throughout a large chip, the advantage becomes very obvious. Raising the level of abstraction of design is the only way to wrap our brains around larger and larger designs, and reducing the lines of code required is a key aspect of raising the level of abstraction and improving design productivity. SV interfaces and structs can help do this by

collapsing large blocks of extremely tedious and error-prone code into a much more compact representation.

Netlisting tools can help by creating the connections, and by providing compact code to read. But they are expanded when seen by all standard tools, and so debug must be done on the expanded versions. Interfaces allow the compact notation to be continued even when debugging, without requiring the complication of two separate versions of the source for the user to maintain.

III. Design Issues

A. Interfaces on Synthesis Boundaries

One of the biggest problems faced with interfaces were when they were used on synthesis boundaries, where one module receiving an interface through a port is synthesized independently of the module containing the interface, as shown in Figure 1.

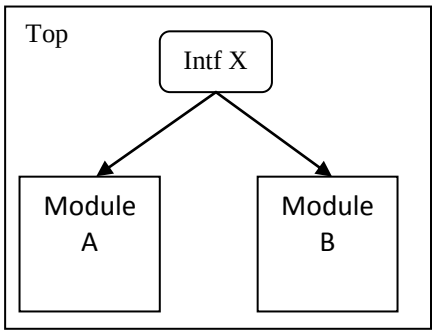

Figure 1: Interface on synthesis boundary

If module A and/or B are synthesized independently of the Top module, then problems arise when module A or B do not use all the signals in the interface, as is often the case. This may occur because the receiving modules use a modport that selects only a subset of the interface's signals. The modport also determines signal direction. The result of the modport selections are not known to the Top module, so the synthesizer will simply include all signals, but module A and B will only include their selected signals. When the synthesized components are later connected, there will be a mismatch for signals and drive direction.

The solution to the problem gives us our first recommendation:

Recommendation 1: Specify the modport at source and destination.

> *When connecting interfaces across synthesis boundaries, you must specify the modport in the connections on that boundary as well as in the receiving module, as shown in Figure 2.*

```
module A(X_if xif.mportA);
…
endmodule

module Top();
   X_if U_if();
   A U_A( .xif(U_if.mportA) );
endmodule
```

Figure 2: Specify modports in netlists as well as input ports.

Unfortunately, referencing a modport in a netlist like this causes problems in other tools that have not accounted for this usage. For this reason, we recommend that you *only* use them on physical design boundaries, rather than making them a standard usage on all interface instances. But we will also discuss how to get around these induced tool issues later in the paper.

A similar issue occurs when parameterizing interfaces on a synthesis boundary. This is really the same problem as parameterizing modules on a synthesis boundary. In both cases, the synthesizer doesn't know what values will be selected when it synthesizes the instanced module. In our flow, we simply disallow parameterized modules on a synthesis boundary. And if we really need one, we just put a wrapper around the module so that it is not longer on the synthesis boundary. But Design Compiler does handle this issue if you tell it directly what the parameter values will be.

If an interface with a different parameter value cannot be avoided, then we have our second recommendation:

Recommendation 2: Duplicate interfaces when specialization required on synthesis boundary.

> *If interface specialization is required on a synthesis boundary, then duplicate the interface and make the necessary change in the new copy. We just changed the default values for the parameters in our new copy.*

This is rather a brute force solution, but the only one we came up with. This normally introduces a problem for verification, but we have a tidy solution for that, which is explained in the Verification section. We also would move the common body of the interface to an include file, then include that in each specialization of the interface. That kept the maintenance hassle to a minimum.

A third problem occurred with interfaces on synthesis boundaries. This came about when a module used only a subset of the signals on a modport. We initially used just full master, slave and monitor modports. But some modules didn't use the clock or reset from the interface, or they didn't use a few of the signals or the full width of some busses. So the solution here is to make liberal use of modports.

Recommendation 3: Add new modports as needed.

> *Add a new modport when one module will not use all the signals in an existing modport. There is no downside to having many modports.*

A slightly more cumbersome problem occurs when you only want to use a slice of the signals in one vector. This occurred

frequently with an address bus, where the master drove the full address, but the slave devices only used a handful of the address LSB's and an external fabric did the MSB decode. SystemVerilog has a handy solution for this problem, called modport expressions. Unfortunately, even our simulator does not support modport expressions, so for now you cannot define a modport that only uses a slice of a vector. So our solution here is to break such vectors up into two pieces such as addr_msb and addr_lsb. Then create a modport for the master with both vectors, and create a modport for the slaves using just addr_lsb.

Recommendation 4: Break signal vectors into common pieces.

When a module requires only a slice of one vector, break the vector into two or more pieces and create modports with the appropriate selection of pieces.

For example:

```
logic [31:0] addr;
logic [7:0] addr_lsb = addr[7:0];

modport master (
  output addr;
  …
);

modport slave (
  input addr_lsb;
  …
);
```

Figure 3: Modport with signal slice

Note that all of the recommendations in this section (recommendations 1 thru 4) are only required for interfaces used on synthesis boundaries. The synthesizer will evaporate any unused signals when it knows which signals both sides are using.

B. Clock Source

We had some difficulties when the clock corresponding to the interface was not available where the interface was instantiated. You must instantiate an interface above the modules that receive it, as shown in Figure 1. You cannot instantiate the interface in one module and then have it output from that module to be connected to the receiving module. So if the clock is only available in one or both of the receiving modules, how do you connect that up?

The typical interface defines a clock and reset as ports to the interface, and all other signals are defined as signals internal to the interface. But if the clock or reset signals are not available where the interface is instantiated, then this usage breaks down. Even if the RTL logic does not need to receive the clock from the interface, the interface still needs to have a clock and reset for verification purposes. You could just add a few ports to your modules and pass the signals up, but this solution is often not acceptable.

So the solution here is fairly simple. Just remove the clock/reset signals from the interface port and make them internal signals just like all the others. We also used our simulation-only ifdef to make these signals evaporate for synthesis.

Recommendation 5: Move clock and reset signals where most easily driven.

Move clock and reset signals off the interface ports when they are only accessible in the source or destination module using the interface.

An alternative option here is to have the clock and reset signals defined both as ports and as internal interface signals. Then just define all the modports required to select the appropriate set of signals or use a parameter driven generate block to select the desired clock input. This eliminates the need to possibly have multiple interfaces defined for the same bus. We did not try this option in our project, but it seems worth considering.

C. Interface Definition

We found that we tended to occasionally put too many signals into a single interface. This just made more work to deal with the unused signals on synthesis boundaries, and also made the design intent a little less clear than desired.

Recommendation 6: Avoid putting too much into a single interface.

Use multiple interfaces when large sets of signals are often unused in one instance.

D. Structs

Some of our designers found structs to be very useful. We particularly found them handy for creating a module of control/status registers. We would typically create a single struct for each register in a unit. Then create a single struct that contained all the other structs. This made for very readable and debuggable code. See figure 4 for an example of the structs package.

```
package regs_pkg;

  typedef struct packed {
    logic [31:24] rsvd;
    logic [23:16] id;
    logic [15:0] count;
  } regA_t;

  typedef struct packed {
    logic [31:0] data;
  } regB_t;

  typedef struct packed {
    regA_t regA;
    regB_t regB;
  } unitA_regs_t;

endpackage
```

Figure 4: Structs Package

We then placed the entire struct as a single output port on the register module as shown in figure 5. We also used the full struct as an input port on any module needing access to the registers. Some designers did not like the full set of registers as an input to their module, so they just peeled off individual registers from the struct and fed them in on their own ports. We could have grouped sets of signals with unions, but no one felt this was necessary.

```
import regs_pkg::*;
module regfile(
  input clk,
  …
  output unitA_regs_t regs
);

always @(posedge clk) begin
  …
  regs.regA.count <= regs.regA.count + 1;
  regs.regB.data    <= data;
end

endmodule
```

Figure 5: Register Module with struct output

We didn't have any problems with structs, except when unused signals in the struct crossed a synthesis boundary. It that case we just had to peel off only what we needed.

IV. Tool Issues

A. Interface Synthesis

We used Design Compiler (DC) from Synopsys for synthesis. The main problems we ran into were due to unused signals that cut across synthesis boundaries, as discussed in the previous sections. Most of those problems where handled by how we coded the interfaces and their connections.

But we also had a similar issue when we used an abstract model. In Figure 1, consider module A as an abstract model that is synthesized separately. Then the Top module is synthesized with module A referenced as an abstract model. But then the abstract model has had its interface expanded into many ports, which need to be mapped to the as yet unexpanded interface in Top. DC needs a little help with this. All you need to do is include the following line in your DC setup when synthesizing Top:

set hdlin_enable_elaborate_ref_linking true

We also ran into problems because interface signals are referenced as 'intf.signal'. DC does not like the period notation in the signal name, so we replaced all period separators with underscores by adding the following line to our DC setup file:

change_name –rules verilog –hier

We also discovered another solution to the problem that occurs if you pass in a parameterized interface on a synthesis boundary. The enclosing module knows the parameters passed to the interface, but the receiving module does not know the parameter values for the interface. So when the two modules are synthesized separately, they will have different size signals and will cause an error when mated back together.

This problem can be solved relatively easily by creating a dummy wrapper module that instantiates the interface and receiving module. The interface instance should be parameterized as it is in the real enclosing module. Then, when you synthesize the receiving module, analyze and elaborate the wrapper module as well, then remove the wrapper with remove_design. The wrapper is thus only used to tell DC the parameter values of the interface that is used by the receiving module. You can find more details on this solution in reference 4.

B. Struct Synthesis

When we used structs or typedefs in module ports, synthesis failed because the package defining these types had not yet been imported. This was no problem for simulation, but required that we move the import statement outside the module for synthesis.

Recommendation 7: Declare package import statements outside module definition.

If structs defined in the package are used as ports on the module, then import that package above the module definition. This is also true for typedefs used as module ports. See example in figure 5: Register Module with struct output.

And you'll also need to make sure DC uses the proper SystemVerilog parser with the following DC setup option:

set hdlin_sverilog_std 2009

C. UPF Synthesis

We had one problem with interfaces when synthesizing our power requirements defined with a UPF definition. This problem occurred because DC required us to replace 'intf.signal' with 'intf_signal', as described previously. But this

then caused a mismatch with the period notation used in the UPF file. We resolved this problem by maintaining two separate versions of the UPF file – one with periods and the second with the periods replaced by underscores. We believe we can avoid this duplication by reading in the UPF file before the change_name directive is applied. However, we did not get an opportunity to try this out before our tapeout.

D. Simulation

The only problems we had with simulation were with our UPF power simulations. We used VCS and MVSIM from Synopsys for power simulations. We ran into three separate issues, but were able to work around all of them and filed bugs with Synopsys for all three. As of this writing, these bugs still exist in the most recent release of VCS.

The first problem occurred when we tried to specify isolation policies for a module that had an interface port. MVSIM could not handle the interface reference in the UPF file, so we had to specify explicit isolation policies for all signals in the interface. Having MVSIM automatically expand the interface is not only convenient, but also helps us avoid a missed signal if we were to ever add a signal to an interface. With the automatic insertion of isolation buffers for the interface, we had to remember to add them whenever we added a new signal. While this is not a serious problem, it does require another pre-tapeout audit.

The second problem occurred when we isolated an interface output that was fanned out from an interface input, then tried to simulate with the isolation turned off. This uncovered a bug in MVSIM that caused the output signal to not be driven correctly. We had to simply add an intermediate signal between the interface input and interface output, as shown below in figure 6.

```
module foo(in_if aif, out_if bif);

// this works
  assign
    enable = aif.enable,
    bif.enable = enable;

// this does not work
  assign bif.enable = aif.enable;
endmodule
```

Figure 6: Use intermediate signals for MVSIM

Note that the problem only occurs when the input signal is fanned out to multiple output signals. So the example above actually works, but will fail if there are multiple copies of out_if.

The final simulation problem with MVSIM was due to the source modport references in our netlists, which were required to handle interfaces on synthesis boundaries as discussed previously. Unfortunately, MVSIM could not understand this use of modports. Our only solution here was to ifdef the netlist connections, such that we did our power simulations without the modport reference and did our synthesis with them. This was an unpleasant work-around, but our only option at the time.

E. DFT

We had problems with our MBist tool when it ran across SV interfaces. We used Tessent MemoryBist from Mentor. We had two basic problems. First, the original code parsers could not handle SystemVerilog interfaces at all. So we updated to the newer HDLE parser, which was able to parse SV interfaces.

However, like MVSIM, the HDLE parser could not understand our use of source modport references in the netlist. It would give a confusing error, indicating there were 'too many connections' when we instantiated a module with an SV interface port. To work around this, we used Mentor pragmas around the definition of the interface port in the module, and around the connection to that interface port in the netlist. The tool would interpret the pragmas as identifying non-synthesizable code, and so ignored it. For example:

```
unitX U_dut(
  .clk(clk),
//mbist_etassemble translate_off
    .xif(U_if.mportA),
// mbist_etassemble translate_on
  …
```

Figure 7: MBist pragma

V. DEBUG AIDS

Once we had interfaces in our RTL, we found several handy uses for them to aid with debug.

As already mentioned, dragging an entire interface into a wave viewer was a very handy option. Not only was it a quick way to get all the signals from one bus into the wave viewer, but we were guaranteed to get all the correct signals for the bus, with no stragglers. We also defined some simple TCL scripts for our wave viewer that we used to display the signals in the desired order and format. For our larger busses, this was a real time saver. We'd like to see a feature in our waveform viewers that would map a TCL script or other formatting definition to each interface type. Then the formatting would be automatically applied whenever the corresponding interface is displayed. But for now, we just call the scripts manually.

With interfaces we can also display more than just the proper signals on a bus. We added additional signals to the interface that were assembled from other signals. This was very useful to help decode the current state of the bus. One very useful example was to have a signal that represented the number of outstanding transactions on the bus. Having many transactions outstanding on a bus can make it difficult to associate an address phase with a data phase, so a few simple debug signals can make this process much easier. Each time we pulled in an interface, we got all the supporting signals automatically. All our DV signals were placed under an `ifdef that evaporated them for synthesis.

We also added a simple instance identifier to our interfaces by adding the following code:

```
logic [1023:0] instance_path;
initial instance_path = $sformatf("%m");
```

You could then look at the value of this variable to determine where the interface was instanced, regardless of where you accessed the interface. This was useful because the wave viewers we used would only display where the interface was referenced from, which was often not where it was instanced. This is just an anomaly of interfaces. If you pull in an interface from some module that receives it as a port, then the instance shown will be the path to that module. But if an interface was routed through many levels in your design, you often want to know if two interfaces actually represent the same set of signals. By looking at the instance_path we defined, we could easily determine if they were. This was a very nice feature compared to using just wires, as you could easily determine that the busses at two points in the design were on the same wires and be guaranteed that there could be no wiring errors - because interfaces are not 'wired', they are passed as entire objects.

Having an instance reference to the interfaces is a great timesaver because there is no longer a need to look at two sets of wires from two points in the design and try to determine if they were the same bus or not. We would like wave viewers to automatically provide this capability.

VI. INTEGRATION WITH VIP

The VIP must be able to use the RTL interfaces to drive and sample signals. However, a problem arises when the RTL interface is parameterized, which makes it difficult to use by the VIP. Several publications can be found detailing why parameterized interfaces do not work well with VIP and ultimately it's best to avoid them altogether. One could impose a restriction to avoid using parameterized interfaces in RTL, however that is a high price to pay since parameterized interfaces are essential in RTL. There is another solution which we describe in this section. We can create a separate unparameterized interface for the VIP which we will embed inside the parameterized RTL interface. The embedded interface is shared with the VIP using the UVM resource database. The following sections will describe how to integrate the DV interface while minimizing the complexity of the connections to the testbench.

A. *Constructing the DV interface*

The first step is to create a separate interface construct for use by the VIP as shown in figure 8. This will be known as the "DV interface".

```
`define MAX_WIDTH 64
interface dv_intf (
  input clk,
  input rst_n,
  inout valid_dv,
  inout [`MAX_WIDTH-1:0] data_dv,
  inout ack_dv
);
  clocking mcb @(posedge clk);
    default input #(10ps) output #(10ps);
    output data_dv;
    output valid_dv;
    input  ack_dv;
  endclocking
  modport master_mp (clocking mcb);

  clocking scb @(posedge clk);
    default input #(10ps) output #(10ps);
    input  data_dv;
    input  valid_dv;
    output ack_dv;
  endclocking
  modport slave_mp (clocking scb);

  clocking wcb @(posedge clk);
    default input #(10ps) output #(10ps);
    input  data_dv;
    input  valid_dv;
    input  ack_dv;
  endclocking
  modport monitor_mp (clocking wcb);
endinterface
```

Figure 8: Simple DV interface

The DV interface is not parameterized, and starts with the concept of the "maximum footprint" interface discussed in [2]. This means all bus widths are set to the maximum allowable, and the VIP uses bit slices of these buses.

Interface bus signals are often excluded from the port list and instead declared as logic types inside the interface itself. However, declaring the interface signals as ports into the interface as shown in Figure 8 gives two benefits:

1. Flexibility to allow connection to RTL which does not use a SystemVerilog interface using a verilog "bind" and hence minimizing cross module references for that case.

2. Compactness of using the ".*" implicit port connect notation when embedded in the RTL interface.

The ports must be bi-directional to allow connection to either a slave or master driver. The suffix "_dv" is used for bus signal names to differentiate them from the RTL signal names for reasons which will be apparent in the next section.

B. *Constructing the RTL Interface*

A simple RTL interface is shown in figure 9:

```
interface rtl_intf
#(
    parameter BW = 32,  //BUS WIDTH
)
(input clk, input rst_n);

  logic  [BW-1:0] data;
  logic  valid;
  logic  ack;

  modport master_mp (
    output data,
    output valid,
    input  ack
  );
  modport slave_mp (
    input data,
    input valid,
    output  ack
  );
  modport monitor_mp (
    input  data,
    input  valid,
    input  ack
  );

…

endinterface
```

Figure 9: Simple RTL Interface

The RTL interface is completed with an embedded DV interface as shown in Figure 10. The DV interface is hidden from synthesis using a compiler directive. Typically most verification environments maintain such a switch so using it here does not come at any extra cost.

```
interface rtl_intf
#(
    parameter BW = 32,  //BUS WIDTH
    parameter DV_DRVR = 0,
    parameter DV_MSTR = 0,
)
(input clk, input rst_n);
…

`ifdef TBBUILD
  wire valid_dv;
  wire [BW-1:0]data_dv;
  wire ack_dv;
  // instantiate DV Interface
  dv_intf dv_if(.*);

  generate
   if (DV_DRVR == 1) begin
     if (DV_MSTR == 1) begin
       //DV Master Driver to slave RTL
       assign data   = data_dv[BW-1:0];
       assign valid  = valid_dv;
       assign ack_dv = ack;
     end else begin
       //DV Slave Driver to master RTL
       assign data_dv  = {`MAX_WIDTH'hz,data};
       assign valid_dv = valid;
       assign ack      = ack_dv;
     end
   end else begin
     //RTL-to-RTL driver, DV monitor only
     assign data_dv  = {`MAX_WIDTH'hz,data};
     assign valid_dv = valid;
     assign ack_dv   = ack;
   end
  endgenerate
`endif
endinterface
```

Figure 10: Embedded DV interface

The connections to the DV interface are made inside a generate block to manage the direction of assignment. There are three possibilities that result in different signal assignments:

1. A verification component is a master BFM actively driving the RTL slave bus.
2. The RTL is a master driving a verification component actively responding as a slave BFM.
3. An RTL master bus is driving another RTL slave bus, and VIP acts as a passive monitor.

Two parameters are defined in the RTL interface to configure the assignment direction. The default values are set to allow RTL driving RTL, in order to avoid configuring verification infrastructure in the design itself. If the VIP is a master BFM, the DV_DRVR and DV_MSTR are both set to 1. If the VIP is a slave BFM, the DV_DRVR is set to 1 and the DV_MSTR is set to 0.

Wires are required to connect the RTL logic signals to the DV interface ports since the logic type will not connect directly to the inouts. By keeping the wire names the same as the DV signal names, an implicit port connection can be made using ".*".

Note that the modports defined in the RTL interface are not associated with any clocking block. The interface modports are required to connect RTL to RTL, however synthesis tools do not support clocking blocks here.

C. *Connecting to the environment*

The DV interface must be shared with the VIP. The first step in doing this is to pass the interface to the UVM resource database as a virtual interface construct so it is visible to the verification environment. This is done inside the RTL interface itself:

```
`ifdef TBBUILD
  import uvm_pkg::*;

  initial begin
    uvm_resource_db
    #(virtual dv_intf)::set(
    "interfaces",
    $sformatf("%m.dv_vif"),dv_if);
  end

`endif
```

Figure 11: Environment connection

Considering there may be several such buses, the name given to the resource database must be unique yet predictable. This is achieved using the verilog "%m" to create a name string based on the hierarchical path of the DV interface.

In the case of an RTL-to-RTL connection, the RTL interface is already instantiated inside the DUT, and the DV interface appears in the resource database without additional testbench code. In the case of a DV master or slave driver, the RTL interface must be instantiated in the testbench and connected to the RTL.

The testbench then pulls the virtual interface from the resource database by recreating the name string, and pushes it to the correct agent using the UVM config database.

```
initial begin
  #0
  if (!uvm_resource_db
    #(virtual dv_if)::read_by_name("interfaces",
    $sformatf("%m.U_dut.rtl_if.dv_vif"),dv_vif))
      `uvm_fatal("TB ERROR","VIF not found")
  else begin
    uvm_config_db
    #(virtual dv_if)::set(
        null,
        "uvm_test_top.env.dv_agt",
        "dv_vif",
        dv_vif);
  end
end
```

Figure 12: Testbench connection

The read from the resource database follows a "#0" which is used to avoid the race between pushing the DV interface and reading it, both of which occur inside initial blocks.

The agent can now pull the virtual interface from the config db as shown in Figure 13:

```
uvm_config_db #(virtual dv_if)::get(
   this,
   "",
   "dv_vif",
   dv_vif);
```

Figure 13: Agent connection

The steps taken to simply pass a DV interface to its agent may at first seem excessive; however, their justification is best explained when one considers other simpler solutions which do not work as well. Two of these are described below.

1. The agent can pull the interface directly from the resource database and cut out the testbench as a middle man. However, this requires the agent to have knowledge of the RTL hierarchy which should be avoided in an OOP environment. Furthermore, multiple interfaces of the same type create different UVM names, and the agent or environment must somehow manage which virtual interface belongs to which instance of the agent.
2. The testbench can find the DV interface in the DUT hierarchy instead of in the resource database. This would eliminate the need for the RTL interface to push the DV virtual interface to the resource database. This solution is better than the first one but it has a drawback since it requires a hierarchical path into the RTL. Cross-module references should be avoided whenever possible. The solution proposed in this paper avoids the cross-module reference by hiding the hierarchical path inside the UVM name string.

VII. THIRD PARTY IP

One of the key impediments to the use of interfaces is their lack of availability from third party IP providers. We would like to see an interface option available from our purchased IP. This could be facilitated by having a standard interface definition for all standard busses. We would encourage the standards bodies to consider developing such standard interface definitions.

But even if we did have a standard interface definition for commonly used busses, there is still the problem of interface specialization. Any standard interface must be parameterized to be of general use, but this causes problems if the interface is used on a synthesis boundary. And all the other problems that we've discussed that require a custom version of an interface would also preclude the use of a standard interface.

What is needed is some sort of base definition for an interface which can be used by a receiving module, but can then be extended somehow to allow the necessary customizations but still be compatible with the standard interface used by IP. Bromley and Vreugdenhil [3] provide an elegant solution with their proposal for 'abstract modports'. The idea is that you define a standalone modport which provides all the definition necessary for a module receiving the modport, but then can be referenced in any interface which can later be passed into the receiving module. Bromley and Vreugdenhil give the following example:

```
package pkg;
  modport Abstract (
    output logic [3:0] L );
endpackage

interface AI;
  import pkg::*;
  logic [7:0] Vec;
  Abstract a_mp(.L(Vec[5:2]));
endinterface

module CE3 (pkg::Abstract P(.L(v)));
  initial v = 4'b0;
endmodule
```

Figure 14: Abstract modport example

Then we only need standardized abstract modports, rather than a standard for a complete interface. Each user can then define their own interfaces, to meet their particular needs, but can still easily pass in their interface to standard IP. The user interfaces can have many additional modports, even additional signals and assertions, etc., but they only need to also include the standard modport used on the IP to ensure that it will correctly connect and not leave any dangling signals that could cause some of tool issues we have discussed.

VIII. Proposed Improvements

1. Tool support for modport expressions.

   A modport expression is a defined SystemVerilog construct, but it is not well supported. From section 25.5.4 of the SystemVerilog LRM [1]: 'A modport expression allows elements of arrays and structures, concatenations of elements, and assignment pattern expressions of elements declared in an interface to be included in a modport list'. For example:

```
interface I;
  logic [7:0] r;
  const int x=1;
  bit R;
  modport A (output .P(r[3:0]), input .Q(x), R);
  modport B (output .P(r[7:4]), input .Q(2), R);
endinterface
```

Figure 15: Modport expressions

   The example shows how you can select just part of a vector to be included in a modport. This eliminates unused signals from being included in a modport or the need to create new signals to accomplish the same thing, as we described in recommendation 4.

2. LRM extension for abstract modports.

   See discussion under section VII: Third Party IP.

3. Automatic instance reference for interfaces in wave viewers.

   See discussion under section V: Debug Aids.

4. Automatic mapping between interface types and formatting script.

   See discussion under section V: Debug Aids.

5. Interface Navigator in Wave Viewers.

   When you debug an issue in waves, you will typically want to look at selected interfaces first. If you've gone to the trouble of including interfaces in your design, it would be very handy if you could see a nice hierarchy of just your interfaces so you could quickly navigate to the ones of interest. Having a few pre-defined wave sets just doesn't scale well to a large design with a hundred or more key interfaces. We'd like to see an option in a wave viewer's hierarchy navigator that would directly highlight all the available interfaces, with filtering by interface type.

6. Ability to reference a parameterized virtual interface without specifying the parameters. This would make it easier to reuse the same RTL interface for DV purposes.

7. Ability for an interface to reference itself, similar to the "this" pointer in a class. This would allow any interface to push itself to the UVM resource database, without requiring a wrapper. Combined with a solution for #6, this would eliminate the need for a separate DV interface in most cases.

IX. What You Can Do to Help

The main reason interfaces are not better supported by industry tools is because the vendors are not pressed to make them work. An easy way to solve that is to include just a single interface in your project. If you ever get to a point where you run into a blocking issue, it's quite trivial to remove that single interface. But you can submit the issue to your tool vendor and press them to get you a fix. If each project did the same with a single interface, or other SV features with limited support, then the vendors would very quickly get these issues ironed out.

X. Conclusions

Support for SV interfaces and structs in RTL design has come a long way since interfaces were first defined by Accelera back in 2003. Yes, they are 10 years old now. We had some problems with them on our project, but we found work-arounds for all the issues and have pressed our vendors for better solutions going forward. Our primary problem was using interfaces across synthesis boundaries. The solution for that problem – referencing modports in the module connection – proved troublesome for some other tools. But we were able to successfully tapeout our chip using a modest number of interfaces and structs. We had designers that liked using them and some that did not. Our verification did appreciate their use, and they helped considerably with debug. The authors believe that more extensive use of interface and structs would make their benefits seem more valuable, especially as tool support improves.

Acknowledgment

We'd like to thank Allan Carter, Sasi Murugesan and Adam Tate for their diligence and persistence to resolve problems encountered with SystemVerilog interfaces and structs in our design process. We owe them our undying gratitude. Or at least a round of beers.

REFERENCES

[1] IEEE (2012) "Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Language", *IEEE Std 1800-2012*.

[2] Aron Pratt, "Parameterized Interfaces and Reusable VIP", VIP Central, September 25, 2012

[3] Jonathan Bromley, Gordon Vreugdenhil, "Is There a Future for SystemVerilog Interfaces", *Proceedings of DVCon 2009*.

[4] Synopsys, "Building SystemVerilog Designs Using a Bottom-Up Approach", Solvnet, 2013.

[5] Stuart Sutherland, Don Mills, "Synthesizing SystemVerilog. Busting the Myth that SystemVerilog is only for Verification", SNUG Silicon Valley 2013.