Towards a Practical Design Methodology with SystemVerilog Interfaces and Modports

Jonathan Bromley
Doulos Ltd
Ringwood, U.K.
jonathan.bromley@doulos.com

***Abstract*—Explores the benefits and limitations of SystemVerilog interfaces and modports in block-level design. Identifies key problems of portability, re-use and flexibility in interface-based design, and suggests a methodology for adoption of SystemVerilog interfaces and modports that helps to solve these problems in synthesizable designs.**

I. INTRODUCTION

The *interface* construct provided in the SystemVerilog hardware design and verification language [1] offers a means to encapsulate complicated interconnect, just as Verilog's *module* construct encapsulates functionality. However, interconnects used in typical modern electronic systems and subsystems are not merely wiring. They contain non-trivial functionality within the interconnect structure itself (often known as the *bus fabric*), and subsystems connected to such a bus fabric need logic to support the often complex protocols required by interconnect standards. It is not obvious how the *interface* construct can be used to support all common interconnect modeling styles.

Section II reviews the key features of, and motivation for, SystemVerilog's *interface* construct, and shows how it can be used to model interconnect structures.

Section III describes what the author believes to be the key challenge associated with the use of interfaces in design, and shows that the most obvious solutions are unsatisfactory in practice.

Section IV examines more radical solutions to the problem discussed in section III, and argues that these solutions are out of reach until tool vendors provide more complete SystemVerilog language support than is available at present.

Sections V and VI consider how the advanced features of interfaces can be used to make designs more expressive, and address some concerns about robustness and ease of use.

Finally, section VII concludes with some specific design methodology recommendations that respect the limitations of current tools, and summarizes changes to tools and language features that could make interfaces more widely useful.

II. INTERCONNECT MODELING USING SYSTEMVERILOG

A. *Data structures are not enough*

Like most other programming and hardware-description languages, SystemVerilog offers a rich set of constructs for modeling user-specified data structures. In particular, it has the *struct* and *union* construct and the useful concept of *packed* data objects. *Packed* data in SystemVerilog are stored in contiguous collections of bits having a well-defined organization, so that (in design applications) the mapping between data structures and the underlying hardware that carries them is explicit. However, these packed structures are inappropriate for modeling complicated interconnect structures. They do not capture information about direction of signal flow, nor do they allow for the specification of different views of the interconnect for different kinds of client modules that may connect to it. Both these issues are elegantly answered by the *interface* and *modport* constructs in SystemVerilog. Note that, in the remainder of this paper, the word "interface" is used exclusively to refer to the language construct in SystemVerilog and not in its ordinary usage.

B. *Interconnect modeling using SystemVerilog interfaces*

The SystemVerilog interface construct is closely similar to a Verilog or SystemVerilog module both in the syntax of its declaration and the language constructs it is permitted to contain.[1] It has, however, two special distinguishing features:

- it can contain *modport* constructs;
- an interface instance may be connected to a module's port, giving that module access through the port to the interface's entire contents but appearing as a single connection in the module's instantiation.

---

[1] Reference [1] forbids module instances within an interface. However, this restriction is not enforced by all tools and is likely to be lifted in a future revision of the standard.

```
interface Sig_Intf;
  logic Sig;
  modport driver_mp (output Sig);
  modport receiver_mp (input Sig);
endinterface

module Driver(Sig_Intf.driver_mp si);
  initial si.Sig = 1'b1;
endmodule

module Receiver(Sig_Intf.receiver_mp ri);
  initial #1 $display(ri.Sig);
endmodule

module Top_Fig1;
  Sig_Intf S ();
  Driver   D (S.driver_mp);
  Receiver R (S.receiver_mp);
endmodule
```

Figure 1. Interface with modports connecting two modules

A *modport* defines a view of an interface, specializing the interface so that a specific kind of client module can connect to it in an appropriate manner. Fig.1 illustrates two modules, one a driver and the other a receiver, connected together by a very simple interface containing only one signal. Modports `driver_mp` and `receiver_mp` capture views of the interface as required by driver and receiver modules respectively. The modules explicitly choose to connect to the modport appropriate to their behavior. It is useful to note that dataflow direction in a modport is specified from the point of view of the connected module rather than from the point of view of the interface. Thus, for example, in our `driver_mp` modport, signal `Sig` is specified as an output from a connected driver module.

For a trivial interface as shown in Fig.1 there is obviously little benefit in the use of interfaces. Indeed, it adds some complication to the connected modules – note, for example, the "dotted name" `ri.Sig` that must be used in the receiver module. However, if the interface encapsulates large and complex connectivity, the simplification achieved in the enclosing module is very valuable. In particular, if the interface represents a standard bus structure that is replicated in more than one place in the design, bus signals within the interface can retain their standard names, since they are encapsulated in an interface instance. It is not necessary to invent names for signals in the various instances of the bus structure.

C. *Key features of this modeling style*

Superficially, the use of interfaces described in the previous paragraphs is a straightforward extension of the Verilog language. It has always been possible to capture a large collection of interconnect in a module and instantiate that into an enclosing module. An interface merely makes it more convenient for us to reach into that "module" by allowing it to be referenced through a port of a module that wishes to connect to it. However, there are two far-reaching repercussions of this change.

- The port connection in each connected module makes reference not to a variable or net in the enclosing module, but to an *instance*. We have, in effect, bound an identifier (the module's port name) to an existing hierarchical instance. This ability is a radical addition to the Verilog language. SystemVerilog extends this notion yet further by providing a new kind of variable (a *virtual interface*) that can be bound to an interface or modport instance dynamically, at run-time, although this feature is inappropriate for synthesis and is not discussed further here.
- The usage of interfaces and modports shown in Fig,1 is synthesizable by at least two commercially available tools.[2] (The Verilog code shown in the connected modules is simplified for the sake of brevity and is not synthesizable, but if it were, then the whole design would be synthesizable.) The important change here is that synthesis tools have traditionally refused to accept any kind of hierarchical name or cross-module reference but, in the special case of members of a connected interface, synthesis of forms such as `ri.Sig` is possible. Current tools achieve this by flattening each interface instance so that it becomes a collection of signals declared in the instantiating module, and then rewriting all connected modules' port lists appropriately.

III. THE PROBLEM OF MULTIPLE DRIVERS

The style exemplified in Fig.1 is perfectly suited to modeling a system structure in which each module is "plugged-in" to a backplane or similar bus structure having a number of identical sockets or attachment points all wired in parallel. Each modport represents one possible kind of socket. There is no limit on the number of modules that can connect to a modport (although, as discussed later, a mechanism exists in SystemVerilog interfaces to restrict a modport so that only zero or one module instance may connect to it). All module instances connecting to a given modport will see exactly the same set of interconnect, with the same directionality attributes.

A. *Three-state drivers*

The approach described above works very well for multi-drop bus structures in which each module has a driver on its outputs but only one module's drivers are active at any given time. In such a scheme, the output drivers of currently inactive modules must refrain from driving their outputs, typically by asserting a high-impedance ("Z") value onto those outputs. Multi-drop buses of this type are common and

[2] It is likely that there are further synthesis tools, of which the author is unaware, that offer similar capability.

```
interface TS_Intf;
  wire Data;  // uses a net
  logic [7:0] Adrs;
  modport
    master_mp(output Data, input Adrs);
  modport
    slave_mp (input Data, output Adrs);
endinterface

module TS_Master(TS_Intf.master_mp mi);
  initial begin
    mi.Adrs = 50;  // select first source
    #100
    mi.Adrs = 42;  // select other source
  end
endmodule

module TS_Slave
  #(parameter A = 0)
  (TS_Intf.slave_mp si, input logic data);
  assign si.Data = (si.Adrs == A)
                 ? data   // selected
                 : 1'bz;  // deselected
endmodule

module Top_Fig2(input logic d42, d50);
  TS_Intf   T ();
  TS_Master M (T.master_mp);
  TS_Slave #50 S50 (T.slave_mp, d50);
  TS_Slave #42 S42 (T.slave_mp, d42);
endmodule
```

Figure 2. Multiple slaves using three-state drivers

```
interface Var_Intf;
  logic Data;  // uses a variable
  logic [7:0] Adrs;
  modport
    master_mp(output Data, input Adrs);
  modport
    slave_mp (input Data, output Adrs);
endinterface

module Var_Master(Var_Intf.master_mp mi);
  initial begin
    mi.Adrs = 50;  // select first source
    #100
    mi.Adrs = 42;  // select other source
  end
endmodule

module Var_Slave
  #(parameter A = 0)
  (Var_Intf.slave_mp si, input logic data);
  always @*
    if (si.Adrs == A)  // when selected...
      si.Data = data;  // ...update result
endmodule

module Top_Fig3(input logic d42, d50);
  Var_Intf   V ();
  Var_Master M (V.master_mp);
  Var_Slave #50 S50 (V.slave_mp, d50);
  Var_Slave #42 S42 (V.slave_mp, d42);
endmodule
```

Figure 3. Multiple slaves using selective write to variable

appropriate for systems built at the level of a circuit board or a rack-and-cards system, but are usually inappropriate at the on-chip level where three-state drivers (those capable of asserting a high-impedance output) present many difficulties relating to testability and other issues.

Fig.2 shows a simple example in which the master module provides an address signal to the interface through a dedicated modport `master_mp`. Each slave module, connected via modport `slave_mp`, compares the address signal with a parameter value. If its address matches, a slave assumes it has been activated and it drives a value on to the interface's common `Data` net. Inactive slaves place a high-impedance value on the `Data` net.

Some synthesis tools, notably those targeting FPGA devices that lack on-chip three-state drivers, can automatically restructure such a description so that a set of three-state drivers is mapped on to a multiplexer with equivalent functionality. However, this facility is not universally available; and in any event the author is uneasy about a technique demanding not only that the designer specify an apparently inappropriate architecture, but also that the synthesis tool then convert it to a very different architecture.

B. *Selective writing to variables*

As an alternative to three-state drivers on a multi-drop signal modeled as a net, the signal could be modeled as a variable. Only the currently active connected module writes to that variable; other connected modules are deselected and refrain from writing to the variable. This approach works well in simulation but is unlikely to be synthesizable, because it requires the synthesis tool to resolve Verilog's "last write wins" semantics across assignments from code in multiple module instances. Most synthesis tools cannot do this resolution even for the much simpler case of assignments from multiple processes in the same module.

Fig.3 modifies the example of Fig.2 to show how multiple data sources can be implemented using selective write to a variable in the interface, as described in the preceding paragraph. It is probably appropriate to emphasize once again that this approach simulates correctly but is not synthesizable with currently available tools.

IV. PROPOSED SOLUTIONS

A. *Selection (addressing) functionality in the interface*

SystemVerilog interfaces can include functionality, much in the same way as modules. Consequently it would be straightforward to add address-decoding functionality to the

```
interface Arr_Intf;
  logic Data;
  logic slave_D[0:1];
  logic [7:0] Adrs;
  modport
    master_mp(output Data, input Adrs);
  modport
    slave_mp (input slave_D, output Adrs);
  always_comb
    Data = slave_D[0] | slave_D[1];
endinterface

module Arr_Master(Arr_Intf.master_mp mi);
  initial begin
    mi.Adrs = 50;  // select first source
    #100
    mi.Adrs = 42;  // select other source
  end
endmodule

module Arr_Slave
  #(parameter A = 0, slave_ID = 0)
  (Arr_Intf.slave_mp si, input logic data);
  always @*
    if (si.Adrs == A)  // when selected...
      si.slave_D[slave_ID] = data;
    else               // not selected
      si.slave_D[slave_ID] = 0;
endmodule

module Top_Fig4(input logic d42, d50);
  Arr_Intf   A ();
  Arr_Master M (A.master_mp);
  Arr_Slave #(50, 0) S50 (A.slave_mp, d50);
  Arr_Slave #(42, 1) S42 (A.slave_mp, d42);
endmodule
```

Figure 4. Interface with slave signal array

interface, thus locating it in the bus fabric (the interface itself) rather than in the connected modules. Each slave module has its own distinct modport, with its own dedicated data signal. Selection logic in the interface drives the appropriate data signal on to the interface's common `Data` signal.

Whilst this approach is straightforward, and readily implemented using today's tools, it has the severe drawback that the bus fabric must be given a distinct modport for each connected module. Consequently the interface definition cannot be generic and re-usable. Worse still, each modport needs a distinct data signal, and these distinct names are visible to the connected slaves. As a result, each slave must be specialized for the modport to which it should connect. We do not consider this approach to be useful in practice.

B. An array of signals, one per slave

The approach of section IV.A can be usefully improved by providing an array in the interface, as shown in Fig.4. The whole of this array can then be made visible to all connected modules through a single modport. Each connected module is parameterized for the array subscript that it will use, and takes care to write only to that element of the array. Within the interface, address decoding or other selection logic determines which element of the array is to be copied to the common `Data` signal; in our example we have ORed together the various signals.

Whilst this style is readily implemented with current tools, and can be used to model most typical bus structures, it seems clumsy. It is awkward to parameterize, and requires careful coordination of parameter values on the interface and on its connected modules.

The design described in [2] uses an extended form of this technique in which calls into functions in the interface are also parameterized for the slave identity.

C. Point-to-point interfaces; with bus fabric in a module

The bus fabric, together with module selection (address decoding) functionality, can be implemented in a traditional Verilog module. SystemVerilog interfaces can then be used to capture the point-to-point interconnect between each connected module and its dedicated port on the bus fabric.[3] Some features of interfaces remain valuable even in this restricted use model, as illustrated in Fig.5. This style has a number of important benefits.

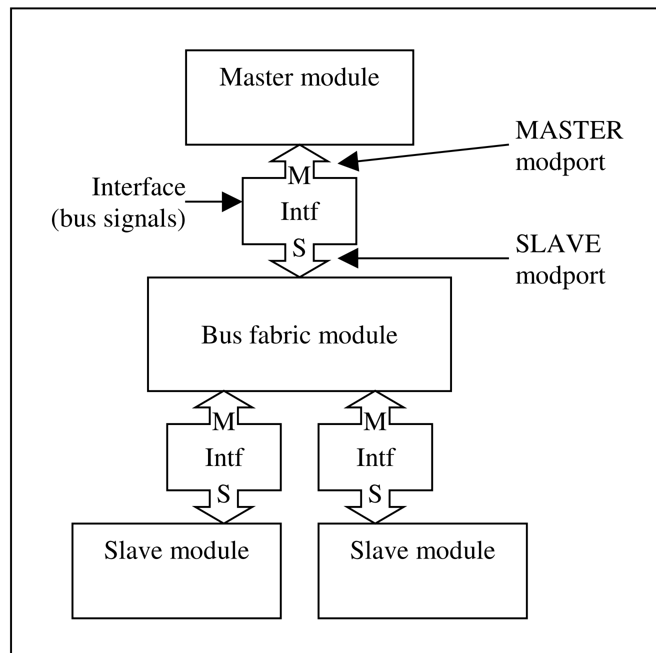

Figure 5. Bus fabric module and point-to-point interfaces

[3] The author is grateful to B. Mathewson of ARM Ltd for bringing this technique to his attention.

- Each point-to-point interconnect uses the same set of signals. An interface allows the standard names of these signals to be used even when there are multiple instances of the interconnect at the same level of the design hierarchy (as will surely be the case when using this approach).
- Modports are valuable to distinguish the two ends of a point-to-point interconnect. For example, connected slave modules will see the bus fabric as a master; connected master modules will see the bus fabric as a slave. If it possesses modports for both slave and master, the interface used to model each point-to-point interconnect can correctly reflect this distinction in a uniform manner.

This usage idiom is a good fit with typical modern multi-level bus architectures, and presents no difficulties for current simulation and synthesis tools. It makes use of the interface construct in a rather straightforward manner, and pushes most of the design challenges into the bus fabric module. In practice, the bus fabric module is likely to be customized by a specialized software tool to suit the user's system requirements, and therefore questions of re-usability of the bus fabric module do not apply: re-use is achieved through the customization tool.

D. *Generated modports with modport expressions*

Earlier in this section it was noted that it is troublesome to implement address decoding in the interface because each connected module then needs a distinct modport of its own. Not only is this clumsy, it is also error-prone because there is no straightforward way to forbid multiple modules from connecting to a given modport. (This issue of *singleton modports* will be discussed in a later section).

SystemVerilog supports *modport expressions*, in which a modport not only specifies which signals in an interface are visible, but also provides alias names for some or all of these signals so that the connected module sees the alias name rather than the signal's real name. This feature can usefully be combined with the *generate* construct to build an array of modports, all identical from the point of view of their connected modules, but making use of different signals within the interface itself. Fig.6 shows an example of this technique. It shows several key features of the proposed modeling style:

- The form `interface.modport_name` used in each slave module's port list allows the connected modport to be specified without specifying which interface defines it. This facility makes it possible for more than one interface implementation to provide the same modport façade, readily allowing progressive refinement of the bus fabric design without disturbing the design of connected modules.

```
interface Gen_Intf
  #(parameter N_slaves = 2);

  logic [7:0] Adr;
  logic slave_D   [0:N_slaves-1];
  logic slave_sel [0:N_slaves-1];
  logic Data;

  always_comb begin : selector
    int slave_ID;
    case (Adr)  // Address decode
      42: slave_ID = 0;
      50: slave_ID = 1;
    endcase
    slave_sel = 0;
    slave_sel[slave_ID] = 1'b1;
    Data = slave_D[slave_ID];
  end : selector

  modport master_mp (
    output Adr,
    input Data
  );

  generate
    genvar i;
    for (i=0; i<N_slaves; i++)
    begin : Slave
      modport slave_mp (
        input Adr,
        input .sel(slave_sel[i]),
        output .Data(slave_D[i])
      );
    end : Slave
  endgenerate

endinterface : Gen_Intf

module Gen_Master(Gen_Intf.master_mp mi);
  initial begin
    mi.Adrs = 50;  // select first slave
    #100
    mi.Adrs = 42;  // select other slave
  end
endmodule : Gen_Master

module Gen_Slave(
  interface.slave_mp si,
  input logic data
);
  assign si.Data = data;
endmodule : Gen_Slave

module Top_Fig6(input logic d42, d50);
  Gen_Intf   G ();
  Gen_Master M (G.master_mp);
  Gen_Slave S50 (G.Slave[0].slave_mp, d50);
  Gen_Slave S42 (G.Slave[1].slave_mp, d42);
endmodule : Top_Fig6
```

Figure 6. Generated modports

- The multiple similar modports are constructed in a `generate` loop. Consequently, it is important to understand the scope names that this `generate` loop creates, so that the correctly scoped modport instance can be hooked by each client module.
- In our example it is the interface (bus fabric) that chooses which slave modport is selected as a function of an address value generated by the master. However, address decoding can instead be performed within the client modules; it is merely necessary for each module to provide a "selected" signal back to the interface.

This modeling style accurately reflects bus fabric structures in the interface. It readily allows the bus fabric to specialize its modport instances – for example, allocating a specific address range to each – whilst making the various modport instances appear identical from the point of view of a module connected to them. The author regards this as the most natural style for using interfaces to represent bus structures in modern designs.

Unfortunately no commercially available simulation or synthesis tools support it, as far as the author is aware. This is both disappointing and surprising, especially as the SystemVerilog language reference manual explicitly provides an example of just such usage of modport expressions.

V. Further Opportunities for the Use of Interfaces

A. *Encapsulation of protocol functionality in an interface*

Interfaces can contain subprograms (tasks and functions) as well as data objects that represent interconnect. These subprograms can be made available to connected modules via the interface's modports using the *import* construct. If such subprograms are functions rather than tasks, and are declared *automatic* so that they do not imply storage elements, then such functions are synthesizable and offer a means for an interface to define functionality that will ultimately be used in a connected module. In particular, an interface could contain a state variable and next-state function defining the state machine that will ultimately be synthesized into a connected module. Since a bus protocol in an RTL design is typically implemented using a state machine, this mechanism offers a way for an interface to contain not only a specification of interconnect but also a specification of the protocol that connected devices should maintain on that interconnect. This technique has been successfully used in a synthesizable design [2] and is supported by several currently available tools.

B. *Verification-related uses*

The use of interfaces as a bridge between testbench and device under verification has been recommended in several published verification methodology documents such as [3] and [4]. In this application, two non-synthesizable SystemVerilog features – *clocking block* and *virtual interface* – are used in conjunction with interfaces to provide a complete, packaged solution to a number of issues relating to configuration and timing of the connection between a testbench and its device-under-verification. This application of interfaces is already well established in verification practice and widely supported by current tools, and it is not considered further here.

C. *Progressive refinement across varying levels of abstraction*

The ability to import and export subprograms through modports of an interface offers some interesting possibilities in simulation environments composed of models some of which operate at a relatively high level of abstraction (commonly known as transaction-level models) and some of which are more concrete, perhaps including timed behavioral models and synthesizable RTL models. Since the current paper's concern is with synthesizable RTL modeling, this possibility is not considered further here.

VI. Robustness and Encapsulation

The approaches outlined in section IV make it possible to create designs in which interconnect is hidden or encapsulated in an interface. Whilst this is highly desirable and can represent a significant benefit for RTL designers, there are some further opportunities for improving the robustness and packaging of such designs that are worthy of note. Some of these are, at least in principle, supported by the SystemVerilog language in its current form.

A. *Singleton modports*

By default, modports of a SystemVerilog interface are promiscuous. There is no limit to the number of modules that can connect to a given modport instance. Whilst this facilitates parallel connection of numerous modules to the same set of signals, it is entirely inappropriate when a modport has been specialized to the needs of a specific, single connected module (as in several of the styles shown in section IV). It would be preferable to restrict such a modport so that it can have at most one connected module, and it seems appropriate to describe such a modport as a *singleton*.[4]

This singleton requirement can be met in at least four different ways, described in the following sections.

1) *Exported function*

A modport can include a function *export* construct, giving the interface access to a function declared inside a connected module. Unless the *extern forkjoin* specification is used (as described in clause 20.8.4 of [1]), such an export construct becomes illegal if more than one module connects to the modport. Regrettably, the *export* construct is not supported for synthesis and as a result this technique is

[4] Various dictionaries offer *chaste* as an antonym of *promiscuous*. The author regards *singleton* as more apposite in the current context.

currently unavailable. As indicated in section VI.B below, support for this feature would not only offer singleton modports but would also enable a useful additional design style.

2) *Drive a uwire*

The Verilog-2005 standard [5] introduced a new kind of net known as a *uwire*. Nets of this new kind are permitted to have at most one driver. If an interface contains a net of the *uwire* kind, and that net is driven by a module connected to one of the interface's modports, then there can be only one such module. Unfortunately, the author is aware of only one simulator, and no synthesis tools at all, that handle the *uwire* construct at the time of writing.

3) *Write to a variable*

As already mentioned, synthesis tools permit only one process to write to any given variable. Consequently, if a variable exists in an interface and is written by a connected module through a modport, there can be at most one such connected module. However, the SystemVerilog language permits any number of processes to write to a variable. The singleton property therefore cannot be enforced in simulation but merely becomes a synthesis restriction.

4) *Programmer discipline*

The aforementioned three ways to implement singleton modports are all unavailable in practice. Solutions (1) and (2) depend on language features that are not currently supported by synthesis tools, and (3) is unsatisfactory because the restriction is not enforced in simulation. As far as the author is aware, it is in practice only by careful coding that a modport can be treated as a singleton. This is usually fairly straightforward, but it is disappointing that it must be left to users to enforce this behavior for themselves.

B. *Address decoding functionality in a connected module*

Most of the styles described in section IV implement a readback multiplexer as part of the bus fabric interface. Such a multiplexer must be selected by means of an address decoder of some kind, and the obvious location for this address decoder is in the interface itself. However, this approach means that the interface must be specialized or parameterized for the address range of each connected module.

An alternative, mirroring the approach taken by many bus-oriented hardware systems, is to perform address decoding in the connected modules. Individual modules can then be parameterized for address range as part of the configuration of a top-level design, and the bus fabric remains generic. However, readback multiplexers must nevertheless be located in the interface rather than the connected modules. Activation of the select signals for such multiplexers could, perhaps, be determined by the results of calling an address-matching function supplied by each connected module. This is infeasible with current tools, because it requires use of the *export* construct in the same way as described above in connection with singleton modports. However, the effect can readily be mimicked by providing an output from the connected module to convey the result of the decoding function back into the interface.

VII. CONCLUSIONS

Whilst the uses of SystemVerilog's interface construct for verification are already established, there is little agreement on the most appropriate ways to use it in RTL design. Of the approaches suggested in this paper, the most promising (the use of modport expressions in modports within a generate construct) is not supported by current synthesis and simulation tools. It is much to be hoped that this situation will improve in the near future. Some interesting possibilities for parameterization of bus-based designs could in principle be supported by the use of already-standardized features of interfaces, but limitations of current tools (and most especially the absence of synthesis support for functions exported from modules into an interface) mean that many of these possibilities are currently out of reach for RTL designers.

A. *Limitations of the language*

Section VI.A shows that there is no satisfactory way, with current tool support, to enforce singleton behavior on a modport. Even if tool support were available, the singleton behavior would be a side-effect of other language features. The author regards it as a deficiency of the SystemVerilog language that it offers no explicit means to restrict or enforce the number of connected modules on a modport.

The modport construct provides a means to capture a client's view of an interface. To facilitate progressive refinement, and to enable design of generic modules that can inherit their interconnect protocol from whatever interface they are connected to, it would be preferable for a modport to be a language construct in its own right – not merely a feature of an interface. Because modports can exist only as a component of an interface, designers have only two choices: either to replicate a modport definition in each and every interface that supports that modport, or to enclose the modport in a small interface that is then instantiated in each interface that supports it.

ACKNOWLEDGMENTS

The author is grateful to his employers Doulos Ltd for the time, facilities and encouragement they provided for this work, and to his colleagues for consistently enlightening discussion.

Cliff Cummings acted as reviewer and provided valued feedback.

Many members of the SystemVerilog community have unwittingly provided helpful hints during informal discussions.

REFERENCES

[1] IEEE Std.1800-2005. IEEE Standard for SystemVerilog – Unified Hardware Design, Specification,and Verification Language.

[2] Jensen P, Kruse T, Ecker W, 2004. SystemVerilog in Use: First RTL Synthesis Experiences with Focus on Interfaces. SNUG Europe, 2004.

[3] Bergeron J, Cerny E, Hunter A, Nightingale A. Verification Methodology Manual for SystemVerilog. Springer 2005.

[4] Glasser, M et al. Advanced Verification Methodology Cookbook version 2.0. Mentor Graphics Corporation, Wilsonville, Oregon, 2006.

[5] IEEE Std.1364-2005. IEEE Standard for Verilog Hardware Desicription Language

[6] Sutherland, S. Modeling with SystemVerilog in a Synopsys Synthesis Design Flow Using Leda, VCS, Design Compiler and Formality. SNUG Europe, 2006.