# Is There a Future for SystemVerilog Interfaces?

Jonathan Bromley
Doulos

Gordon Vreugdenhil
Mentor Graphics

The SystemVerilog interface is intended to be a powerful modeling construct for describing hardware interconnect in a very general manner that is applicable to both testbench and synthesizable RTL design applications. In this paper we argue that the SystemVerilog interface construct is inadequately specified, insufficiently powerful for real applications, and impossible to implement consistently in its current form. We then review the application areas that interfaces were intended to address, and propose some possible solutions for these shortcomings.

The SystemVerilog language has had a dramatic impact on the hardware design and verification world since its introduction in 2003 and IEEE standardization in 2005. Notwithstanding the wide and enthusiastic adoption of SystemVerilog among design and verification engineers, there is an interesting and radical feature of the language that has achieved only limited acceptance: the interface construct.

In this paper we use the word *interface* specifically to denote the SystemVerilog language construct of that name, and not in its more general meaning.

First we outline the history and current status of interfaces. Next we offer a number of applications for the interface construct. Some of these applications are in common use today; others are technically possible, but have met with limited popularity; yet others are perhaps desirable but for various reasons are not fully supported by interfaces in their present form. Given the limited use of interfaces that is found in real designs this section is necessarily anecdotal and imprecise, being based on discussion with a number of experienced users and on plausible extrapolations from current practice. Nevertheless, we hope that it sheds light on some areas in which interfaces could be improved.

In the next section, we discuss a number of concerns relating to the current definition, and available implementations, of interfaces. Significant shortcomings are identified both in the language reference manual [1], which lacks precision in this area, and in the usability of interfaces.

We then offer a number of specific proposals for enhancements of the interface construct to make it more usable and to close some loopholes in its definition. The changes suggested there do not claim to be definitive; our motivation is explicitly to provoke discussion in the community of users and implementers, the results of which we hope will inform future revisions of the standard.

## BACKGROUND

### History

The interface construct was implemented in the language Superlog [2], which was based on Verilog. That form of interface was added, almost without alteration, to SystemVerilog [1].

### Tutorial Information

Information on the use of interfaces in RTL (synthesizable) design can be found in [3][4][5]. The use of interfaces in testbench design is described in [6][7][8][9].

Early documentation on interfaces emphasized their ability to model interconnect at a wide variety of levels of abstraction. Although this has not received much attention in practice, descriptions of such usage can be found in [10][11].

### Current Status

Interfaces are supported, at least to some extent, by a wide range of SystemVerilog simulation and synthesis tools. However, robust support for two of the most interesting features of interfaces (modport expressions and the creation of modports inside a generate construct) is available only in a much smaller set of tools.

The use of an interface to encapsulate the set of signals that will be manipulated by a verification component, and the use of virtual interface variables to gain access to such interfaces from dynamically constructed verification component objects, is well established in object-oriented testbench design practice. It has been explicitly promoted by two prominent verification methodology toolkits [8][9] and appears to be very widespread in practice, although an alternative approach that avoids the use of virtual interfaces has also been described in [12].

## APPLICATION AREAS FOR INTERFACES

In this section we outline some of the areas where interfaces seem to be useful. Not all of these applications are widespread in practice, but all have been described and proven possible at least to some level.

### Encapsulation of Interconnect in RTL Design

A commonly mentioned application of interfaces is as a means to encapsulate, in a single design unit, a set of interconnecting signals that can be used to link two or more module instances. All signals in the set are declared within an interface, so that any instance of the interface contains one such set of signals. With a few minor limitations, this usage is synthesizable. It shows some promise in the description of bus-based designs.

Modern interconnect structures are much more than a simple multi-drop bus. They typically contain arbitration, decoding, multiplexing and other functionality, and usually have a dedicated port for each connected client module. It is natural to consider implementing such a structure as a SystemVerilog interface, with one modport for each connected client. However, such usage with interfaces is not straightforward, as we indicate below.

#### Point-to-multipoint

Interfaces in their present form readily support broadcast and multi-drop connection schemes. They are much less well suited to so-called *point to multipoint* connection in which some signals are broadcast from a master module to several client modules, and some signals are returned from each client to the master. In such a topology it is necessary to create multiple connection points that share a common external appearance (analogous to a backplane connector) but are differentiated from the master's point of view. This arrangement is analogous to having some pins on a backplane connector dedicated to each specific connector rather than being bussed to all. As argued in [4], the current facilities of interfaces and modports provide frustratingly poor support for such schemes.

#### Propagation of modports

If a module has a port of modport type then that port can without difficulty be exposed as a port of its enclosing module. It would be very desirable if, conversely, a modport of an interface could be exposed as a modport of an enclosing interface that instances it.

#### Composition of modports

It often arises that an interconnect structure is composed in a hierarchical manner from other, smaller structures. The AXI bus [13] with its five channels is a good example: each channel is somewhat selfcontained and may even be handled at some level by an individual module, but at higher levels of the design hierarchy all five channels must come together to form the complete bus. Although an interface containing five instances of channel interfaces can easily be constructed, there is no way to create a modport that captures a set of other modports. Further work is needed to establish how best to meet this evident requirement.

### Encapsulation of Signals to be Manipulated by a Testbench

The ability of interfaces and modports to capture a set of connections makes them attractive as a way to provide the signal-level interconnection between a testbench and the RTL device-under-test. For dynamically constructed (object-oriented) testbench implementations, the use of a *virtual interface* — a variable that can hold a reference to an interface instance — is convenient and has gained wide acceptance as the standard way to link class-based testbench code to the static module instance hierarchy.

Although this is probably the most commonly encountered application for interfaces today, it remains somewhat unsatisfactory. Concerns about the data type of virtual interfaces, discussed in [15] and later in this paper, can make their use somewhat clumsy, and the lack of any notion of interface inheritance makes it hard to write testbench infrastructure that can handle heterogeneous virtual interfaces in a consistent manner.

### Abstraction Adapter for Mixed Behavioral and RTL Modeling

Modports on an interface can not only expose signals in the interface, but can also expose subprograms (tasks and functions) in the interface for use by a connected module. Furthermore, this subprogram connection can work in both directions so that the interface can gain access to subprograms implemented in a connected module. This facility allows interfaces to be used as abstraction level adapters, typically having two modports, one appropriate for pin-level connection to RTL signals, the other providing transaction-level access by means of subprogram calls. Tasks or functions in the interface itself can perform the work of translating one abstraction level to the other, as described in [10][11].

### Decoupling of Functionality from Connectivity

Interfaces have the ability to expose subprograms to client modules through a modport, and such usage is in many cases synthesizable. Early in the development of SystemVerilog it seemed that this provided an opportunity to create bus-connected devices — typically peripheral devices, memory controllers and the like — that are interface-agnostic, providing the specified peripheral functionality but able to work with any of a range of bus protocols. This could be achieved by implementing part of the bus protocol logic not in the peripheral device but in the interface to which it is connected. Properly implemented, this could allow an unmodified peripheral module to work correctly on for example AHB, Wishbone or proprietary bus structures. Encouraging preliminary results were reported in [5]. Unfortunately, various limitations of the interface construct in its present form, combined with the inherent difficulty of the problem, have meant that this intriguing idea has received little further attention or acceptance.

## INADEQUACIES OF THE CURRENT DEFINITION

At the time of writing, at least twenty distinct issues relating to interfaces remain open in the SystemVerilog bug tracking database maintained by the IEEE language standardization committees [14]. The majority of these issues have received so little attention that they do not yet have proposals for change. Although some of the issues are minor, there is no doubt that the current definition of interfaces has some perceived ambiguities and inadequacies. Some of the more pressing concerns are outlined below.

### Lack of Precision in the Definition of Modports

The high-level intent of modports is clear: a modport provides access to a specific set of items in an interface, possibly with some renaming (modport expressions), so that each different kind of client that may wish to connect to the interface has its own modport. Some properties of modports are indeed well defined. For example, a module connecting to an interface by means of a modport has access only to the interface items specified in the modport; other items in the interface are inaccessible to it.

Beyond this level of description, however, things are much less clear. For example, consider Code Example 1.

```
interface Itf;
  bit V;
  modport MP(output V);
endinterface

module Mod(Itf.MP p);
  always #1 p.V = ~p.V;
endmodule

module CE1_Top;
  Itf itf();
  Mod mod(itf.MP);
endmodule
```

*Code Example 1.*

It is clear that instance `itf` contains a variable `itf.V`. Less obviously, what does `p.V` denote within instance `mod`?

The obvious answer is that it denotes `itf.V`, but this implies that `p.V` is a *reference* to `itf.V`. This is the semantics of a `ref` port rather than an `output` port, rendering directions `ref` and `output` indistinguishable in modports — an interpretation that is unlikely to be useful.

A second interpretation is that `p.V` is indeed a reference to `itf.V`, but is restricted to be write-only. There is some support for this interpretation in [1] through the comment in Clause 20.4 that “The syntax of `interface_name.modport_name reference_name` gives a local name for a hierarchical reference.” This view however would be culturally incompatible with most other forms of `output` direction in Verilog, and would make the use of `p.V` as an expression (as in the `always` procedure in module `Mod`) become surprisingly and inconveniently illegal if the write-only restriction were to be enforced. Furthermore, it is not clear how compatible such an interpretation would be with respect to synthesis assumptions and lack of support for hierarchical references in synthesis.

The interpretation that seems best aligned with the intent of modports, and most likely to be tractable for synthesis, is to regard the modport `output` as implying a continuous assignment to its target `itf.V`. This, however, raises troublesome questions of its own. First, what is the source expression for this continuous assignment? The only plausible candidate is a new, implicit variable named `p.V`, whose data type is the same as the data type of `itf.V`. This implicit variable presumably exists within module instance `mod`, in much the same way that an ANSI-style output port definition can create an internal variable in a module (albeit one that is clearly explicit). Unfortunately, a module port connection is not the only way to access a modport. A virtual interface variable can be set to reference a modport, and can reference different instances at different times in the simulation. It is far-fetched to imagine that the act of binding a virtual interface variable to modport `itf.MP` should dynamically create not only a continuous assignment to `itf.V` but also an implicit variable to act as the source expression for that assignment.

The differences among these three interpretations are not merely academic. The choice of interpretation has an impact on what is legal and illegal for user code to do, particularly if the user writes to `p.V` from more than one process, or connects more than one module instance to the modport. It is therefore unfortunate that this question is not even mentioned in the SystemVerilog language reference manual.

Unfortunately, this question is not easy to resolve. Synthesis concerns clearly require that a module be isolated from its interconnect by something equivalent to continuous assignment across the port boundary, so that each module can be synthesized in isolation. By contrast, it is completely inappropriate for a virtual interface connection to represent a continuous assignment, because that connection may be severed and re-made at any time during the simulation, requiring that not only the continuous assignment but also its source expression be transitory. As a further concern, there is no clear statement in [1] about the meaning of a modport that has no client module connected to it, or that has multiple modules connected to it.

While there are some similar concerns relating to modport `input` items, they are less pressing. Multiple modules can have ports connected to a given modport instance, but at most one modport instance can be connected to a module's port. Consequently, questions of possible multiple or changing drivers on a target variable are much less troublesome for `input` than for `output` items. While it may be preferable for the sake of consistency to specify that such an `input` item represents continuous assignment across the port boundary, it is probable that the effects of such a decision are invisible to user code.

## Modports and the Provides/Requires Contract

A module port of interface type can be defined in any of four ways, as shown in the port list of the following imaginary module header in which `ITF` is the name of some interface and `MP` is the name of some modport:

### Interface and modport both fully specified

Port `p1` of Code Example 2 must be connected to modport `MP` in an instance of interface `ITF`.

### Interface specified, no modport

Port `p2` can be connected to an instance of interface `ITF` in its entirety, with no modport involved. Alternatively, it can be connected to any modport in an instance of `ITF`. In the latter case, module `M` must use `p2` in a manner that is consistent with the limitations of the connected modport.

### Generic interface with modport specified

Port `p3` can be connected to modport `MP` in an instance of any interface that provides such a modport.

### Generic interface without modport specification

Port `p4` can be connected to any interface instance, or to any modport in an instance of any interface. In the latter case, module `M` must use `p4` in a manner that is consistent with the limitations of the connected modport.

```
module M (
  ITF.MP       p1,
  ITF          p2,
  interface.MP p3,
  interface    p4);
```

*Code Example 2.*

The freedom to choose a specified modport in any interface that has such a modport, as exemplified by port `p3` above, is a very valuable feature of this mechanism. However, the mechanism as it stands is intolerably weak. The existence in some interface of a modport whose *name* is `MP` provides no guarantee that it has the properties required by this module. A capricious or careless user could connect port `p3` to an interface having a completely inappropriate modport named `MP`, and the ensuing elaboration-time errors would wrongly appear to be the fault of module `M` as it attempts to manipulate that modport in an inappropriate way.

Conversely, a module can specify in its port list that it expects to connect to an interface with no modport qualification, as in ports `p2` and `p4` in Code Example 2. It remains possible that the actual connection to this port may be qualified with a modport name and therefore may expose the connected interface in a different way than the module expected. Once again, a choice made by the code that instances a module can silently invalidate some assumptions made within the code of that module. It seems bizarre, and a retrograde step in language functionality, that interface ports provide weaker type checking than ordinary ports.

If a module wishes to obtain a guarantee (contract) that a modport to which it is connected will have the desired properties, it must use a fully qualified port like `p1` in our example. However, experience has shown that this is likely to be much too restrictive for realistic applications.

We propose as an enhancement — described later in this paper — that these difficulties, and many others, could be addressed by allowing modports to be defined as a new kind of data type. This change would allow a module to have a contract with its port connection, guaranteeing that the port will be connected to a modport that has an appropriate set of properties to meet the module's needs. It would also facilitate the provision of the same type of modport by many different interfaces, and would allow a modport to appear more than once in a given interface.

### The Data Type of a Virtual Interface

Reference [15] discusses some difficulties with the current definition of virtual interfaces and outlines the rationale for, and consequences of, some changes that are expected to appear in the forthcoming revision of the language [16]. Although the concerns are rather different, the conclusion is quite closely related to that of the previous section: there is a pressing need for references to an interface or modport to have a known data type, defined independently of the interface instance that will be referenced.

It is important to note that there is a difference in level of abstraction between virtual interfaces and interface ports. Interface ports are permitted to be more abstract by using generic interface connections, are permitted to reference modports by way of a generic interface, and are not required to provide parameterization information in the port type. Virtual interfaces are required to be much more precise in providing the *matching typ*e for the actual interface assignment. The fundamental reason for this difference is that interface port types form part of the elaborated instance hierarchy of the design. The resulting types and legality of the interconnect use are similar to other forms of elaboration resolution of types and hierarchical references. Virtual interfaces however are associated with interface instances during simulation time. In general, it is not possible to determine what associations will occur and then solve the resulting type relationships. Thus in order to allow generic virtual interfaces a system would effectively have to embed an interpretive engine to deal with portions of the design interacting with the generic virtual interface. The performance impact of such approaches would be sufficiently egregious that the standards committees did not believe that the additional generality would be supported by vendors.

### Extensibility of Interfaces and Modports by Inheritance or by Composition

It is not unusual to find examples of interconnect structures having features that are present only in certain instances. This might be because the interconnect structure was defined to have optional features, or because an enhanced or extended version is created based on an existing specification. The AMBA APB peripheral bus structure [17] provides a simple example: it introduced two new signals `PREADY` and `PSLVERR` in the most recent APB3 version of the specification as an optional, backward-compatible enhancement of the earlier APB2 specification.

Given the present definition of interfaces and modports, it would be necessary in a heterogeneous system to create two different interfaces, each with its own modports, to represent APB2 and APB3 bus structures. Given that APB3 is a strict superset of APB2, this problem could be neatly solved if there were some means to create an APB3 interface or modport that is an extension of an existing APB2 interface/modport definition. One might imagine this being achieved by something akin to class inheritance, whereby a new interface definition inherits all the contents (or, at least, all the external appearance) of an existing one. Alternatively, an extension style based on composition might be possible.

At present the only form of extension available for interfaces is to create child interface instances inside another interface. This is not a satisfactory solution, because the modports and contents of the child instance must now be referenced by hierarchical name and there is no provision for modports of a child interface instance to be exposed as all or part of a modport on the enclosing parent interface.

Today, users are forced to replicate appropriate parts of the source code of an interface in the new, extended interface definition.

Of all the changes and concerns mentioned in this paper, extensibility of interfaces and modports has been the most frequently raised when the authors have discussed these matters with experienced users. Some possible approaches towards a solution are outlined in our proposed enhancements, but the problem has some troublesome aspects and further work is needed before a viable solution can be proposed.

### Modport Items Cannot Have Simple Local Names in a Connected Module

When a module has a port of interface or modport type, items in the connected interface must of necessity be referenced using a two-part dotted name of the form *portname.itemname*. Although this presents no fundamental obstacle, it is irksome if these port items make many appearances in a large module. Furthermore, it is an obstacle to easy conversion of legacy Verilog modules to use an interface port. A useful side-effect of the changes proposed in the next section would be the ability to rename modport items so that they have a local name within the module.

## PROPOSED ENHANCEMENTS

The weaknesses outlined in the previous section can be categorized in four major areas:

- poor precision in the language reference manual;
- insufficiently strong notion of the data type of an interface or modport;
- poor usability because some features are absent;
- lack of extensibility of interfaces or modports, either by inheritance or by composition.

In the remainder of this section we suggest some changes to the current definition. The authors wish to emphasize that these proposals are somewhat speculative at present. Our primary goal is to stimulate clear-headed discussion of the problems and possible solutions with a view to introducing significant improvements to the language at the earliest opportunity.

### Language Specification

The current LRM text on interfaces depends heavily on the use of simple examples to illustrate language constructs. Although helpful for new readers, this approach leaves too many ambiguities for comfort. We hope that any future revision of SystemVerilog will encompass rewriting of the relevant LRM clause so that it is more rigorous and complete.

### Data Type of Interfaces and Modports

As has been noted already [15], interfaces and modports have a split personality. They have a type-like nature, which comes into play when creating virtual interface variables and interface-type ports on a module; but the details of that type are determined not by the interface's definition, but by the characteristics of a specific interface instance. We present here a possible solution to the problems posed by this split.

### The contract between a module and its ports

Ordinary (non-interface) ports on a Verilog module have an unambiguous data type that is unaffected by the instantiation of that module (except, of course, that the module may be parameterized and its parameters may affect its ports in a well-understood way). This makes it possible to reason about the correctness of the module in isolation. As an example, consider the following module header:

```
module M1 #(N = 4, type T = int)
   (input pkg::t P1,
    output T P2,
    inout [ N-1:0]  P3);
```

Although we do not know what the values of parameters `T` and `N` will be when this module is instanced, we have enough information to reason about the correctness of the module with respect to its ports. Each port has a data type that is well known in terms of the module's parameters or, in the case of port `P1`, the contents of a package. Each port represents a contract with the module: the port will be connected to a signal of compatible data type (the precise rules for type compatibility depend on the port direction, but are clearly specified).

By contrast, the data type of an interface or modport can be entirely unknown to the module that uses it. Consider a module that has a port of interface/modport type:

```
module M2 (interface.MP p);
```

Port `p` on this module must connect to a modport `MP` in some interface instance, but there is no contract with the module. Suppose the design also includes instances of the following two interfaces:

```
interface I1;
  logic L;
  modport MP(output L);
endinterface

interface I2;
  struct { int x; int y;}  s;
  modport MP(input s);
endinterface
```

The correctness or otherwise of module `M2` depends on whether its port `p` is connected to an instance of `I1` or of `I2`. Nothing in the module's parameterization makes this difference visible to the module. The existence of the interface port provides no guarantees to the module, and imposes no obligations on the parent module that instances it. Given that interfaces are intended to *raise* the level of abstraction of RTL designs, this lack of contract at a module's port boundary is surprising and unpleasant.

### Abstract modports to express the contract

We believe that creating a new kind of data type, the *abstract modport*, to represent an interface façade can solve this problem. An abstract modport should define the data types, directions and names of the items in a modport. Once defined, an abstract modport could be used as the data type of a module's port, where it would offer a contract with the module that the port will be connected to a modport or interface that provides at least the defined items. In an instance of such a module, the port would represent an obligation to connect to an appropriate interface or modport instance. Code Example 3 sketches how this might work, and illustrates some of the features we envisage.

The abstract modport `Abstract` is declared in a package so that its declaration can easily be made available to other design units. Its declaration is superficially similar to a normal modport except that the data type of each item is explicit, and the modport's declaration is not within an interface.

```
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
```

*Code Example 3.*

Interface `AI` is defined in the normal way, but its modport `a_mp` is defined as an instance of `Abstract`, whose declaration is visible thanks to the package import. The obligation imposed by each modport item is satisfied by connecting it to an item of appropriate data type in the interface using a modport expression.

Module `CE5` has a port of `Abstract` type, explicitly taken from the package. Modport item `L` is associated with a local name `v` in the module, using a syntax reminiscent of modport expressions or explicit port declaration. Local name v now denotes the modport item `P.L` in the module.

Because the abstract modport has a clearly defined type, it becomes possible to reason about the module's correctness without needing to know what interface instance has been connected to its port. Other interfaces, quite different from our example `AI`, could safely be connected to this module — provided that they contain a modport of the correct type to satisfy the obligation imposed by its port.

### Backward compatibility

An early draft exposition of this proposal can be found at [18]. It contains detailed discussion of how the proposed new mechanism could be made fully backward compatible with, and could co-exist with, the existing arrangements.

### Parameterization of abstract modports

Rather like parameterized classes, abstract modports should be parameterizable so that a single definition can provide a family of different types of abstract modport. As with parameterized classes, each unique specialization of the abstract modport would create a distinct data type, incompatible with all other specializations.

Separate parameterization of abstract modports as types also alleviates a common issue with the current model for parameterized virtual interfaces. The use of parameterized virtual interface variables in classes generally causes a parameter “cascade” effect in which the interface parameters must be included in the parameters of the class simply to be able to describe the type of the virtual interface.

### Virtual interfaces

Abstract modports capture the data type of the access to an interface that is represented by a modport. It is precisely such access to an interface that is required of a virtual interface variable. It should be possible to declare such variables as `virtual` *abstract_mp_type* as an alternative to the current `virtual` *interface_name.modport_name* form. A virtual interface variable of abstract modport type could then reference only an interface instance that has a modport of exactly the correct abstract type. Again, this change makes it easy to reason about code that uses the virtual interface variable without needing to know about the interface instance that it references.

Using the modport type approach requires that interfaces used with a virtual interface must provide such a modport. However it would be fairly simple to extend the concept of a typed modport to an abstract interface type — an interface could be declared as virtual in a package and such an abstract interface type would be used only to form a contract. Code Example 4 shows how this approach might apply. The example presumes a new keyword `provides` which is semantically similar to class inheritance keyword `extends`. However, the intent here is not to describe actual inheritance of data members, etc. but rather to assert that the actual interface provides at least the data members, modports, etc. defined by the abstract interface.

```
package pkg;
  virtual interface Abstract;
      logic [ 3:0]  L;
  endinterface
  class C;
 virtual Abstract myVintf;
  endclass
endpackage
interface intf provides Abstract;
  logic [ 3:0]  L;
  bit extras;
endinterface
```

*Code Example 4.*

The basic approach in an abstract interface is that any compatible actual interface would have to conform to the “virtual interface” declaration in that the actual interface would have to provide at least the same names and types as the abstract interface. This makes clear the contract expectations between the testbench and the static instance hierarchy, and allows a clean separation between the type and instance aspects of interfaces.

## Feature Enhancements

In this section we outline some usability enhancements that follow from the proposed abstract modports, and suggest a further improvement (parameters as modport items) that has obvious applications in RTL design.

#### Local names for modport items

As suggested in Code Example 3, a module port of modport type could optionally have a connection list in which local names are provided for each modport item. This is intended to remove one obvious barrier to adoption of interfaces and modports, the awkward dotted-name syntax that must currently be used to denote a modport item within a connected module.

#### Multiple similar modports in an interface

Abstract modports make it straightforward to have multiple modports of the same kind in a given interface. It is clear that this is important if an interface is used to create complex interconnect structures. For example, bus structures with numerous attached modules will typically have a distinct port for each module. Many of these ports will have the same characteristics and could be represented by instances of the same modport.

Today this is possible either by writing multiple modports, each with a different name, or by using a generate construct. Copy-and-paste is obviously unsatisfactory. Generate constructs have the disadvantage that they hide the generated modports inside a new named hierarchical scope, making connection more complicated and possibly rendering the design intractable for synthesis.

Using abstract modports the problem is much simpler. Multiple named modports of the same type could be written, or perhaps the instance array mechanism could be used to create an array of modports without forming a new hierarchical scope.

#### Parameters as modport items

Considering again the problem of representing a bus fabric block with multiple ports, we note that each port on such a block must be characterized for details such as its address range, number of byte lanes, interrupt priority and so forth. The attached design blocks may need to be parameterized to match some of these characteristics. Instead of providing the same parameter values in two places to characterize both the bus port and the connected module, it would be attractive to supply the parameter value in just one place and have it propagate automatically into the other as needed. It seems likely that this could be achieved by adding parameters as a new kind of modport item — an elaboration-time constant that is visible to a connected module. Code Example 5 shows a worked example of this suggestion. Abstract modport `Addr_Param` exposes not only the signal ports `Addr` and `Data` but also a parameter `Base`, which will be provided by the interface and used within a connected module. Module `Responder`, designed to connect to this modport, expects to obtain parameter `Base` from the modport. Interface `AD_Bus` provides two instances of modport `Addr_Param` with different values for parameter `Base`, giving different behaviors to the two module instances `R0` and `R1` that are connected to it in enclosing module `CE4`.

### Extensibility of Modports and Interfaces

There are two primary facets to extensibility of modports and interfaces — usability and implementation. Usability is obviously the end goal, but the realities of modern systems require careful consideration of the implementation impact. There are deep historical foundations to Verilog and SystemVerilog that allow implementations to make type and referencing decisions no later than the end of elaboration. Thus simulation can continue using efficient compiled forms which are key to

```
package p4;
  modport Addr_Param
    (parameter [ 7:0]  Base,
     input logic [ 15:0]  Addr,
     output logic [ 7:0]  Data);
endpackage

module Responder (
  p4::Addr_Param ap, ... );
  import p4::*;
  always_comb
    if (ap.Addr[ 15:8]  == ap.Base)
      ap.Data = Response_Data;
    else
      ap.Data = '0;
endmodule

interface AD_Bus (...);
  import p4::*;
  logic [ 7:0]  D0, D1;
  logic [ 15:0]  A;
  Addr_Param mp0 (
    .Base(8'h00), .Addr(A), .Data(D0) );
  Addr_Param mp1 (
    .Base(8'h04), .Addr(A), .Data(D1) );
  ...
endinterface

module CE4 (...);
  AD_Bus ad_bus(...);
  Responder R0(.ap(ad_bus.mp0));
  Responder R1(.ap(ad_bus.mp1));
  ...
endmodule
```

*Code Example 5.*

the scalability required for modern designs. The primary requirement for extensible forms is that the existence of names and the types of the data references be determinable by the end of elaboration. As noted earlier, since ports are connected statically at elaboration time, allowing the highly abstract generic interface port form for ports is acceptable, while having a comparably abstract form for the dynamic associations implied by virtual interfaces is not permitted. For extension, any eventual approach must be constrained in the same manner.

Anecdotal evidence from users whom the authors have consulted suggests that there is a real need to be able to create variables of a generic virtual interface type that can dynamically hold references to interfaces of various different types. For example, this would facilitate the creation an array of base-type virtual interfaces that could hold references to all the diverse interface instances used in a testbench. It also seems clear that RTL designers could benefit from the ability to create a family of related abstract interface or modport declarations, analogous to a class hierarchy in object-oriented programming. The authors do not underestimate the difficulties that this presents. Interfaces and modports form a bidirectional contract between the providing interface and the requiring (client) module, with obligations on both sides thanks to the existence of output items and exported subprograms. Consequently it is not obvious that a derived interface (or modport) will invariably be an acceptable substitute for a base interface.

The simplest view of extension would be based on the abstract interface concept introduced earlier. Any enhanced interface that provides the items in the abstract interface could certainly substitute for the base interface. Although this mechanism would not provide inheritance of the implementation of a base interface, it is not clear to the authors whether interface implementations would be sufficiently reusable to make the complexities of such an approach worthwhile.

## SUMMARY

We believe that the interface construct holds considerable promise both for RTL design and for verification applications. However, we are frustrated by its failure to deliver on that promise, by its poor match of features to real user requirements, and by the inadequate LRM definition that renders it unattractive to users and impossible to implement consistently. We are confident that the SystemVerilog community can rectify all these defects, and we urge that users, language specifiers and implementers give serious attention to that project.

## REFERENCES

[1] IEEE Std.1800-2005 *IEEE Standard for SystemVerilog*. IEEE 2005.

[2] Synopsys Inc. *Superlog Language Definition version K1*, October 2002.

[3] Davidmann S, Flake P, Sutherland S. *SystemVerilog for Design 2nd edition*. ISBN 0 387 33399 1. Springer 2006.

[4] Bromley J. *Towards a Practical Design Methodology with SystemVerilog Interfaces and Modports*. DVCon 2007.

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

[6] Bergeron J. *Writing Testbenches using SystemVerilog*. ISBN 0 387 29221 7. Springer 2006.

[7] Spear C. *SystemVerilog for Verification 2nd edition*. ISBN 978 0 387 76529 7. Springer 2008.

[8] Bergeron J, Cerny E, Hunter A, Nightingale A. *Verification Methodology Manual for SystemVerilog*. ISBN 0387-25538-9. Springer 2005.

[9] Cadence Design Systems Inc, Mentor Graphics Inc. *Open Verification Methodology version 2.0.1*. Available at www.ovmworld.org.

[10] Sutherland, S. *Modeling FIFO Communication Channels UsingSystemVerilog Interfaces*. SNUG Boston 2004.

[11] Bromley J. Seamless *Refinement from Transaction Level to RTL Using SystemVerilog Interfaces*. SNUG Munich 2008.

[12] Rich D, Bromley J. *Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches*. DVCon 2008.

[13] AMBA3 AXI Protocol v1.0 Specification. Document number IHI0022B. ARM Ltd 2004.

[14] SystemVerilog bug item 2318, at http://www.eda.org/svdb

[15] Gran A, Vreugdenhil G, Bromley J. *SystemVerilog Virtual Interfaces and Their Impact on Design Verification*. User-To-User conference, Mentor Graphics Inc, San Jose 2008.

[16] IEEE P1800-2009 draft 8, *Draft IEEE Standard for SystemVerilog*.

[17] AMBA3 APB Protocol v1.0 Specification. Document number IHI0024B. ARM Ltd 2004.

[18] SystemVerilog bug item 1861, at http://www.eda.org/svdb

This paper appeared in *Proceedings of DVCon 2009*, San Jose, CA, pp. 248–255.