Using SystemVerilog “Interfaces” as Object-Oriented RTL Modules

Geoff Barnes
Thales Systems Canada
1 Chrysalis Way, Ottawa
Ontario, Canada, K2G 6P9
+1.613.723.7000x3012
Geoffrey.barnes@ca.thalesgroup.com

ABSTRACT
This paper will show how SystemVerilog *interfaces* can be used as objects when writing RTL. While much has already been written about the *interface* and how it can abstract the connection between *modules*, this paper focuses solely on using the *interface* to increase abstraction within *modules*.

Categories and Subject Descriptors
B.5.2 **[Register-Transfer-Level Implementation]:** Design Aids – *hardware description languages*

General Terms
Design, Languages, Verification.

Keywords
Flop – Flip-flop
OO – Object Orientation
RTL – Register-Transfer-Level

1. INTRODUCTION
The typical RTL *module* written in Verilog tends to contain a lot of incidental “boiler plate” code. Boiler plate code is generally undesirable as it is repetitive, uninteresting and wastes precious designer effort. Its presence is in part due to synthesis, as full expressive use of the language is restricted to the synthesizable subset. This often confines the designer to the narrow coding styles that can be reliably inferred into gates. This begs the question: why write something in an abstract way, when there is only one prescribed "way" to write it? Perhaps a more direct, declarative approach is needed. Take the ubiquitous “flip-flop” for example - each register is typically coded in the same repetitive manner, even though they are pervasive in digital design.

Another reason for boiler plate code is the concurrent nature of HDLs. Traditional sub-programs borrowed from the software world, such as “tasks” and “functions”, can naturally capture and represent procedural code. Unfortunately, they cannot be used to capture the important temporal and parallel nature of RTL that is represented by *always* blocks and other concurrent structures.

The *interface* construct provides some unique features which can be exploited to address these issues.

2. BACKGROUND ON INTERFACES
The *interface* construct introduced in the SystemVerilog standard is a very useful addition. As the name implies, its primary purpose is to simplify the wiring between various blocks in a hierarchy. In its simplest form, it acts as a namespace to “bundle” a set of related signals for use in a portlist. This paper assumes the reader is generally familiar with SystemVerilog and the *interface* construct. Information about interfaces is plentiful on the internet and can be found in the SystemVerilog LRM [1].

A less obvious aspect of interfaces is that they can act as “pseudo” *modules*. The *interface* can have a portlist, be parameterized and contain behavioural code, such as *always* blocks. Unlike a true *module* however, the *interface* cannot instantiate other *modules* - limiting it to a “leaf” status in the design hierarchy. The key to our modelling solution is that the *interface* permits the use of hierarchical reference to its internal members. While *modules* support this feature only for simulation, *interfaces* support this feature for both simulation and synthesis.

3. BACKGROUND ON OBJECT ORIENTATION
Exactly what makes a language or a construct “Object Oriented” is often an academic argument that is beyond the scope of this paper. It is generally accepted, however, that at the heart of OO is the ability to create a class – that familiar construct which combines data and its operations. Closely associated with this, is the concept of decoupling a class' internal implementation from its “clients” in the outside world via a defined interface. Most authorities on the subject will summarize the primary features of object-orientation to include[7]:

- Encapsulation – The ability to contain functionality and data within a namespace and control its access.
- Abstraction – The ability to abstract code to an appropriate level to serve the problem at hand.
- Polymorphism – The ability to implement a class or interact with an object while being “type” agnostic.
- Inheritance – The ability to create new class, by extending or modify an existing class.

For our purposes, we'll view OO in a pragmatic light and will consider most important those aspects that can improve our hardware descriptions. While we don't expect the quality of the circuits we create to be better, we strive to make the process behind creating them better. In HDL based design, "better" code could mean more succinct code, less code repetition, improved readability, better reuse, etc. Using examples, we'll apply explore how well interfaces can provide these benefits for RTL.

4. USING INTERFACES AS OBJECTS

4.1 The Basics

This first example is to illustrate the basic coding style that will be used throughout this paper. Figure 1 shows how a simple inverter can be expressed with an *interface*.

```
interface inverter ;
    logic a;
    logic anot;
    always_comb anot = ~a;
endinterface

module top (
   input       some_input
  ,output logic some_output
);

inverter my_inv();

always_comb begin
   my_inv.a      = some_input;
   some_output = my_inv.anot;
end

endmodule
```

Figure 1 : Basic Interface Example

Since this example is so trivial, it is hard to see the value here. Nevertheless, it demonstrates some key features of the *interface*. First, it shows that the *interface* is a namespace. This namespace acts as a user-defined type and can be instantiated much like a class. Secondly, the example shows how the *interface* can encapsulate concurrent behaviour. In this case, an inverter was defined by the *always_comb* block. Thirdly, the example demonstrates how an *interface's* members can be accessed hierarchically. By using the "dot" notation instead of 'ports', the construct's interface is less verbose than that of a *module*. This style is analogous, as least superficially, to using "getters" and "setters". However, it takes more than "dot" notation to make a construct useful.

4.2 Basic Encapsulation

Writing RTL can require a lot of tedium. Many common constructs require the declaration of intermediate wires and regs. This clutters the code and can often obfuscate even the simplest intent. An example of this could be the code required to infer a tristate driver, as shown in Figure 2.

```
module tri_example (  inout io_old_school  );

reg old_school_en;
reg old_school_out;
wire old_school_in;

assign old_school_in = io_old_school;
assign io_old_school = old_school_en ? old_school_out : 'z;

always_comb begin
   old_school_en = <RHS>;
   if (old_school_in) begin
      ...
   end
   old_school_out = <RHS>;
end

endmodule
```

Figure 2 : Inferred Tri-state

We see here that at least five lines of code are required to create the necessary, wires and regs for the tri-state driver. While five lines does not sound excessive, the wire and reg declarations are simply incidental and do not add any value to the *module* description. Consider the excessive clutter if the design calls for many more instances of the tri-state driver.

With an *interface*, however, a cleaner solution is possible. Figure 3 demonstrates how the tri-state driver can be re-coded with an *interface*:

```
// Create interface to capture tri-state logic and
// incidental wires
interface tripin #(parameter type t = logic) (inout pin);
   t in, out;
   logic en;
   assign in = pin;
   assign pin = en ? out : 'z;
endinterface
```

Figure 3 : Tri-state coded as Interface

```
module tri_example ( inout io_new_way );

tripin new_way (io_new_way);

always_comb begin
   new_way.en = <RHS>;
   if (new_way.in) begin
      ...
   end
   new_way.out = <RHS>;
end

endmodule
```

Figure 4 : Tri-state example

The example in Figure 4 shows how the *interface* creates a neater and more structured solution. The multiple lines from the previous example are reduced to one meaningful line as the incidental nets are hidden within the *interface*. Also, there is a side benefit that the code's design intent is explicitly stated.

While we could have just as easily captured the tri-state driver with a traditional *module*, we would still require the incidental net declarations. Accessing a *module* via a port list and wires is far clumsier than the tidier access provided by the *interface*. It is the combination of encapsulation and convenient access to its members that makes the *interface* an appealing construct.

4.3 A More Useful Example

Given that the information age is completely dependent on synchronous sequential logic, we will focus our attention there.

A common way to depict logic gates and registers at the RTL level is the "cloud-flop" paradigm. The "cloud" represents combinatorial logic, and the "box" on the right represents synchronous logic, which is typically some variation of a D flip-flop.

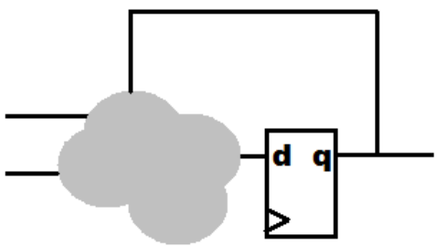

Figure 5 : 'Cloud' to 'Flop' depiction of Sequential Logic

At this level of abstraction, we represent the flip-flop simply as a box. The implementation details of how it 'flips' and 'flops' are not particularly relevant – we simply care that it works as advertised. Yet, when a flip-flop is desired in Verilog, we need to "infer" one and are forced to repeat a piece of code similar to Figure 6.

```
always @(posedge clk or negedge reset_n)
begin
   if (~reset_n) begin
            q <= 0;
   end else begin
            q <= d;
   end
end
```

Figure 6 : Inferred Flip-Flop

It could be argued that this *always* block exposes more low-level details than appropriate for the "Register Transfer" level of abstraction. The interesting parts of the circuit are really the clouds of logic between the flip-flops, as this is where the decisions are made. The flip-flop itself is simply a delay and storage element. Furthermore, the vast majority of the flip-flops in a design are identical in nature – typically positive-edge clocked, asynchronously reset to zero. The *always* block required to infer the flip-flop shows itself as 'boiler-plate' and is a good candidate for encapsulation. Rewritten as an *interface*, the D flip-flop may look like the code shown in Figure 7.

```
interface dflop   #(   type t = logic // deferred type usage
                              ,t reset_value = '0
                   )
   ( input clk,reset,en); // The ports for all our flip-flops

   t q,d; // Familiar d flip-flop naming

   always_ff @(posedge clk or negedge reset) begin
      if (!reset) begin
         q <= reset_value;
      end else if (en) begin
         q <= d;
      end
   end//always

endinterface

// Example : Half-rate Counter
module dflop_usage_example (
                              input clk  ,input reset
                             ,input en   ,output [3:0] foo
                             );

dflop toggle (clk,reset,en); // single flip-flop

dflop #(logic [3:0]) my_counter (clk,reset,en); // 4-bit counter

always_comb begin
   toggle.d = ~toggle.q;
   my_counter.d = my_counter.q;
   if (toggle.q) begin
      my_counter.d = my_counter.q + 1;
   end
end

assign foo = my_counter.q;

endmodule
```

Figure 7 : Synchronous Example

In this example, a simple, reusable, general purpose flip-flop object was created with an *interface*. The basic benefit is that the coding details of the flop's *always* block are contained within the *interface*. The “client” of the flip-flop then need only care about providing the “cloud” feeding the flop's 'd' input. The current state of the flop, is accessed via the 'q' member of the *interface*.

Arguably, this style could be called a “structural” style of RTL coding. Older HDLs, such as ABEL and Altera's AHDL provide primitive constructs not unlike the dflop *interface* above. If nothing else, this style has the benefit of being WYSIWYG (What you see is what you get) . When reviewing this code, for example, it is blatantly clear at declaration time that “toggle” is intended to be a flip-flop.

From an object-oriented point of view, however, the style may more appropriately be considered a “declarative” style of writing RTL. The general benefit is caring only about the “what” and not the “how”. This demonstrates how the *interface* supports abstraction, since it provides a way to encapsulate the “how” details of inferring synchronous logic, and lets us model it at an appropriate level of detail.

This example also demonstrates a useful feature provided by interfaces – parameterization. Like a *module*, an *interface* can be written in a generic way and then later configured with specific values. Parameters in this example permitted each instance to set its own data type, reset value, etc. This may be considered to demonstrate polymorphism since the actual data type of the ‘dflop’ can be deferred, yet the defined operations related to that data need not change.

4.4 Adding Properties for Abstraction

The approach used to create a simple flip-flop, can be extended to do some more interesting things. In synchronous design and in particular when designing control logic, there is often the need to code incidental “utility” flip-flops related to the main control signals. For example, given an input, the designer may want to synchronize the signal, delay it, perform edge detection, and so on. While these operations are important, the details of coding them are largely uninteresting with respect to the main design intent. They are yet another example of 'boiler-plate' code found in RTL. A typical example is shown in Figure 8.

```
// Example : Count falling edges of an asynchronous signal
module example_without_interface (
    input clk
   ,input reset
   ,input async_count_en
   ,output logic [3:0] count_out
);

// Sync and edge detect – uninteresting “utility” code
logic async_count_en_mh1;
logic async_count_en_mh2;
logic sync_count_en;
logic sync_count_en_d1;
logic sync_count_en_falling_edge;

always_ff @(posedge clk or negedge reset)
begin
   if (reset == 0) begin
      async_count_en_mh1 <= 0;
      async_count_en_mh2 <= 0;
      sync_count_en_d1     <= 0;
   end else begin
      async_count_en_mh1 <= async_count_en;
      async_count_en_mh2 <= async_count_en_mh1;
      sync_count_en_d1     <= sync_count_en;
   end
end

assign sync_count_en  = async_count_en_mh2;
assign sync_count_en_falling_edge
            = ~sync_count_en & sync_count_en_d1;


//  Count Edges
logic [3:0] count;

always_ff @(posedge clk or negedge reset)
begin
            if (reset == 0) begin
                        count <= '0;
            end else begin
                        if (sync_count_en_falling_edge) begin
                                    count <= count + 1;
                        end
            end
end

assign count_out = count;

endmodule
```

Figure 8 : Example without Interfaces

Embracing the OO paradigm, we can treat the “utility” flops as metadata - information derived from, or related to the primary concern. In this light, they can be considered as 'properties' for the flip-flop *interface* definition. An example of adding such properties to the dflop *interface* is shown in Figure 9.

```
interface dflop #(
              type t            = logic
             ,t reset_value = '0
             ,async           = 0 // Expect an async input
             )
             ( input clk,reset,en);

t q,d;
t mh1,mh2; // 2 stage hardening flops

// "Attributes" for the flip-flop:
t q1; // follower flip-flop, 1-clock delayed version of flop
t q2; // We could add any number of follower flops if desired
t q3; // Perhaps use a parameter sized array

// Signal will be high when flop goes from low to high
wire rising_edge = q & ~ q1;
// Signal will be high when flop goes from high to low
wire falling_edge = ~q & q1;
// Signal will be high with any change of the flop
wire any_edge = q ^ q1;

always_ff @(posedge clk or negedge reset) begin
   if (!reset) begin
      q   <= reset_value;
      q1  <= '0;
      q2  <= '0;
      q3  <= '0;
      mh1 <= '0;
      mh2 <= '0;
   end else if (en) begin
      mh1 <= d;
      mh2 <= mh1;
       q <= async ? mh2 : d;
       q1 <= q;
       q2 <= q1;
       q3 <= q2;
   end
end//always

endinterface
```

Figure 9 : Interface with properties

```
// Example : Count falling edges of an asynchronous signal
module example_with_interface (
   input clk
  ,input reset
  ,input async_count_en
  ,output logic [3:0] count_out
);

// Sync and Edge Detect - Compact code
dflop #(.async(1)) sync_count_en (clk,reset,1'b1);
   assign sync_count_en.d = async_count_en;

// Count Edges
dflop #(logic [3:0]) count (clk,reset,1);

  always_comb begin
            count.d = count.q;
            if (sync_count_en.falling_edge) begin
                  count.d = count.q + 1;
            end
  end

  assign count_out = count.q;

endmodule
```

Figure 10 : Example using properties

The following properties were added to the flop : q1, q2, q3, rising_edge, falling_edge, any_edge. The properties q1, q2, q3, represent “follower flops” or delayed versions of the main flop. The remaining properties indicate how the signal is switching. These are all automatically maintained by the *interface*, alleviating the need to manually recode these incidental details. Rewriting the previous example using the improved dflop *interface*, the edge counting *module* is now more compact, as shown in Figure 10. This revised example reduces the fifteen or so lines of “utility code” to essentially two concise lines.

Another interesting part of this example is how the *interface* was configured to synchronize an input from another clock domain. The “user” of the flop simply enabled this feature with a parameter and this detail was handled by the *interface*.

4.5 Finite State Machines with Polymorphism

By reusing the basic dflop *interface* from previous examples, just about any synchronous structure can be built. Since the actual data 'type' of the flip flop was deferred through parameters, any arbitrary data type, both standard and user defined may be used. By specifying a user-defined enumeration type, we automatically have abstract state registers for an FSM. This demonstrates the *interface's* basic support for static polymorphism. An example is shown in Figure 11.

```
module fsm_example (
                  input clk
                 ,input reset
                 ,output [1:0] state_out
                  ) ;

   typedef enum logic [1:0] { A , B , C , D} my_states;

   dflop #(.t(my_states) ),.reset_value(D)) st  (clk,reset,1'b1);

   always_comb begin
            st.d = 'x;
      case (st.q)
         A : st.d = B;
         B : st.d = C;
         C : st.d = D;
         D : st.d = A;
      endcase
   end

   assign state_out = st.q;

endmodule
```

Figure 11 : FSM Example

This approach has the added side-effect of enforcing the common coding style of separating the “synchronous” and “combinatorial” always blocks[3]. The synchronous block is hidden in the *interface*, and the combinational block is defined in the calling *module*.

Since FSMs are widely used in designs, it could be useful to create a specialized FSM *interface*. Such an *interface* is shown in Figure 12.

```
// Purpose-built FSM interface
interface FSM
   #(type t = logic, t resets_to = '0)
    ( input clk,reset,en);

t state;
t next_state; // FSM "state" and "next_state" convention

always_ff @(posedge clk or negedge reset) begin
   if (!reset) begin
      state <= resets_to;
   end else if (en) begin
      state <= next_state;
   end
end//always

// Assertions?
// <Your favourite assertions here>

endinterface

module fsm_example2 (input clk, input reset , output logic out) ;

   typedef enum logic [1:0] {A,B,C,D} my_states;

   FSM #(.t(/*my_states*/logic [1:0]),.resets_to(D)) foo
            (clk,reset,1'b1);
// note : Synthesis tool did not accept the 'enum' but
// took the logic equivalent

always_comb begin
   case (foo.state)
      A : foo.next_state = B;
      B : foo.next_state = C;
      C : foo.next_state = D;
      D : foo.next_state = A;
   endcase
end

assign out = foo.state == D;

endmodule
```

Figure 12 : Specialized FSM Example

This version of the FSM *interface* adds a little "syntactic sugar" by renaming the 'd' and 'q' members to 'next' and 'state' as per a common FSM convention[3]. A specialized FSM *interface* would also have the benefit of enhanced readability and could provide a natural place for FSM-specific debug code and assertions.

4.6 Abstracting Clock Domains

In the synchronous examples shown so far, each *interface* used a traditional portlist to define its clock, reset and clock enable signal. This provides an opportunity for further improvement, as we can use an *interface* in its typical role as 'ports'. An example is shown in Figure 13.

```
// Create a wire bundle representing a "clock domain"
interface clock_domain (input clk,reset_in,en);

   logic reset;
   assign reset = reset_in;

   modport domain (input clk, reset_in, en , reset);

endinterface

// dflop recoded to use interface in portlist
interface dflop #(type t = logic,t reset_value = '0)
            (
            interface domain // any interface will do
            );
t q,d;

always_ff @(posedge domain.clk or negedge domain.reset)
begin
   if (!domain.reset) begin
      q <= reset_value;
   end else if (domain.en) begin
      q <= d;
   end
end//always

endinterface

// Example : Synchronize a signal from another clock domain
module example_A (
    input clkA
   ,input areset_sync
   ,input  async_in
   ,output logic syncA
);

// Synchronize the async reset
clock_domain async (clkA,areset_sync,1);

dflop reset_meta (async.domain);
dflop reset_sync (async.domain);
   assign reset_meta.d = 1;
   assign reset_sync.d = reset_meta.q;

// Another domain using the synchronized reset

clock_domain A (clkA,reset_sync.q,1);

// Declare some discrete flops to harden async_in
dflop harden1 (A.domain); // Compact syntax
dflop harden2 (A.domain);
dflop harden3 (A.domain);

always_comb begin
   harden1.d = async_in;
   harden2.d = harden1.q;
   harden3.d = harden2.q;
   syncA = harden3.q;
end

endmodule
```

Figure 13 : Clock Domain Example

In this example, a specialized "clock_domain" *interface* was created by bundling together a clock, a reset and a clock enable signal. Abstracting the clock domain in this way may be useful when writing a block with multiple clocks domains. It allows the design intent to be explicitly stated and provides a single point of control for code maintenance.

In addition to being a simple port bundle, the clock_domain *interface* can be extended to provide extra utility. An example could be the addition of an optional reset synchronizer. The details of the reset synchronizer are hidden from the user, but are conveniently ready to use if required. This is demonstrated in Example 14.

```
// Similar to previous domain, but with an optional reset
// syncer
interface clock_domain #(harden_reset = 0) (input
clk,reset_in,en);

   logic [1:0] reset_mh;

   always_ff @(posedge clk or negedge reset_in)
      if (!reset_in)
            reset_mh <= '0;
      else
            reset_mh <= {reset_mh,1'b1};

   logic reset;
   assign reset = harden_reset ? reset_mh[1] : reset_in;

   modport domain (input clk, reset_in, en , reset);

endinterface

//
// Example : Synchronize a signal from another clock domain
//
module example_B (
     input clkB
    ,input async_reset
    ,input  async_in
    ,output logic syncB
);

// Reset Syncer built-in
clock_domain #(.harden_reset(1)) B (clkB,async_reset,1);

dflop harden1 (B.domain);
dflop harden2 (B.domain);
dflop harden3 (B.domain);

always_comb begin
    harden1.d = async_in;
    harden2.d = harden1.q;
    harden3.d = harden2.q;
    syncB     = harden3.q;
end

endmodule
```

Figure 14 : Domain with built-in reset synchronizer

From an OO point-of-view, this is another example of both abstraction and polymorphism. The clock domain was abstracted by modeling it as a single construct with built-in behaviour. Polymorphism was demonstrated since the dflop *interface* could accept any *interface* as it's "clock_domain". As long as the supplied *interface* supports the 3 referenced members (clk,reset,en), the dflop *interface* is happily agnostic to other details. In this example, the same dflop *interface* accepted a modified variation of the basic 'clock_domain' *interface*.

4.7 Adding Assertions

Encapsulation is not only useful for RTL code, but can also benefit the reuse of assertions. Assertions associated with the RTL in the *interface* can be added to increase the robustness of the design.

For example, an *interface* containing synchronous logic could check for proper synchronous input or unexpected floating input. A specialized *interface* containing synchronizers could check for minimal pulse widths, such as in described in [6]. Refer to Figure 15.

```
interface synchronizer #(
....
property p_stability;
   @(posedge clk)
   !$stable(d) |=> $stable(d) [*2];
endproperty : p_stability


property p_no_glitch;
   logic data;
   @(d)
    (1, data = !d) |=>
   @(posedge clk)
    (d == data);
endproperty : p_no_glitch

assert property(p_stability);
assert property(p_no_glitch);

endinterface
```

Figure 15 : Assertions in interface

A library of well-defined reusable *interfaces* with built-in assertions could prove invaluable by automatically reusing best-practices.

4.8 Code Re-factoring

Abstracting away some of the low-level design details provides an increased opportunity for code re-factoring. In the previous example, internal details of the "dflop" *interface* could easily be changed while not breaking the main code. For example, the *always_ff* block could be recoded to use a different clock edge or a synchronous reset. Doing such a change would typically require manual editing or a sophisticated script to change every *always_ff* block in the design.

Even more extreme re-factoring is possible. An example could be adding "triple modular redundancy" (TMR) to the sequential portions of a circuit, such as in [8]. This technique is used to guard against "single-event-upsets" (SEU) by using redundant logic and voting circuits.

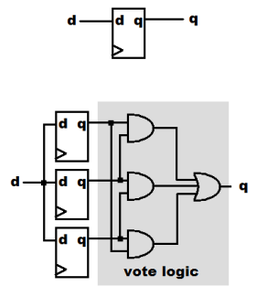

Figure 16 : "Triple Modular Redundant" flip-flop

The "dflop" *interface* can be modified, as in Figure 17, to infer the redundant logic and still be compatible with the previous examples:

```
interface dflop #( type t = logic , t reset_value = '0 )
   ( input clk,reset,en);

t q,d;

genvar i;

localparam TMR = 3;

t vote [0:TMR-1]; //pragma attribute vote preserve_signal true

generate for (i=0;i<TMR;i++) begin : TMR_insertion

always_ff @(posedge clk or negedge reset) begin
   if (!reset) begin
      vote[i] <= reset_value;
   end else if (en) begin
      vote[i] <= d;
   end
end//always

end
endgenerate

assign q = vote[0]&vote[1]  | vote[0]& vote[2] | vote[1]&vote[2];

endinterface
```

Figure 17 : TMR flip-flop

5. BENEFITS OF THIS APPROACH

The preceding examples have shown some of the interesting possibilities provided by the *interface* construct. In particular, we have shown that the *interface* can support at least three features of object-orientation: encapsulation, abstraction and polymorphism. In conjunction, these concepts can provide the following benefits:

- More compact RTL:
  - Encapsulation hides 'incidental' logic.
  - Repetitive concurrent blocks can be encapsulated.
  - Derived logic encapsulated and automatically available.
- More readable RTL:
  - Due to compact code.
  - Abstraction results in explicit design intent.
- Better reuse:
  - Objects can enforce design convention and best practices.
  - Objects can integrate with procedural code.
- Improved Maintenance:
  - Succinct code is inherently easier to maintain
  - Polymorphism and abstraction promote easier re-factoring.

It should be noted that the examples in this paper failed to demonstrate inheritance. This will be addressed in a later section.

5.1 Other Possibilities

SystemVerilog is a substantial language, so there may be many ways to take advantage of the approach shown in this paper.

Aside from assertions, specialized *interfaces* could also be a convenient and natural place to contain debug code. One example could be the jitter emulation technique shown in [6]. This technique involves adding randomization code to emulate metastability in synchronizing flip-flops.

We have focused almost entirely up to this point on the front-end part of the design process, but it may be possible to obtain benefits in the back-end phases as well. By adopting a meaningful naming convention for *interface* members, it is easier to apply wildcard constraints to a netlist. Reusing the example of a synchronizer, the input flop of the synchronizer could be named "async_in". This net name may carry over to the netlist and be able to identify the end of a false path.

6. USING THIS APPROACH

While an entire design could be written using the approach shown in this paper, it is likely not practical to do so at this point in time. The major hurdle is tool support, namely synthesis. Advanced synthesis support for SystemVerilog still appears to be precarious, in the author's opinion, for typical FPGA synthesis tools. While an exhaustive analysis was not carried out, feature support across vendors tends to be inconsistent, making some code non-portable.

Nevertheless, the main examples presented in this paper were synthesizable with a FPGA synthesis tool from a major EDA vendor. Some of the simpler examples were successful even with a FPGA vendor supplied synthesizer. The quality of the results were checked simply by examining the resultant netlist in a netlist viewer. Initial results are encouraging and indicate this approach may be more practical in the future as tools improve.

Note: The example in Figure 12 did have to be modified for synthesis as noted. This appears to be due to an oversight in the tool, and not a fundamental misuse of the language.

Simulation support, as expected, is generally excellent, as SystemVerilog was adopted early for its verification features. It is unknown how other tools such as linters, formal tools, etc would treat this coding style.

7. FUTURE ENHANCEMENT – INHERITENCE

There is currently no way in which an *interface* can be derived from another. It would be useful, however, if the designer could "extend" an existing *interface* by adding or overriding properties, *always* blocks, *functions*, etc. Consider the hypothetical example shown in Figure 18.

```
interface counter ( input clk,reset,en);

   logic [3:0] q,d;

   function void behaviour;
      d = q + 1;
   endfunction

always_ff @(posedge clk or negedge reset) begin : main1
   if (!reset) begin
      q <= 0;
   end else if (en) begin
      behaviour();
      q <= d;
   end
end//always

endinterface

// 'Child' interface #1
interface child1 extends counter; // ports inherited

   // behaviour function overridden
   function void behaviour;
      d = q + 2;
   endfunction

   // named always block inherited
endinterface

// 'Child' interface #2
interface child2 extends counter; // ports inherited

  // named always block overridden
  always_ff @(negedge clk) begin : main1
     behaviour();
     q <= d;
  end

endinterface
```

Figure 18 : Hypothetical Inheritance

In 'child1', we simply replaced the default 'behaviour' with our own version, letting us keep what we wanted to as-is and adjust just one aspect of the *interface* to re-purpose the code. In 'child2', we kept the logical behaviour, we could say, of the circuit, but changed its concurrent behaviour by specifying a new 'main1' always block.

Providing such an enhancement to the language would allow more flexibility and reuse. Inheritance is considered by some to be the key differentiator between object-based design versus proper object-oriented design.[2]

8. REPLACING MODULES WITH INTERFACES

Given that the *interface* is a hierarchy element much like the traditional *module*, we could conceivably create designs based entirely on interfaces. In the simplest form, the *interface* can act as a practical *module* without a portlist. One could argue that the *interface* is essentially a *module* with a flexible instantiation syntax since a port mapping is not required. Port direction is generally not an issue inside the digital core of a design, and as designs get bigger, many *modules* and thousands of lines of RTL spend their entire existence nowhere near an IO buffer! It may be acceptable then to use the *interface* for logical design partitioning, and reserve the *module* for cases where a physical partition is likely. This is consistent with the synthesis tools, which tend to “dissolve” the *interface* during synthesis[4].

However, there is potential for a “hybrid” approach. Since an *interface* has an optional portlist, those signals within the portlist could identify the “hard” ports of a design. As a top level, these would become physical pins. Other nodes, which are essentially 'public' members but not ports, would simply remain as internal nodes. If the block is used as a leaf *module*, then those nodes can be optionally connected as we have shown.

Currently, using *interfaces* instead of *modules* for general purpose hierarchy is not practical. The main limitation is that interfaces cannot instantiate *modules*. Therefore, instantiating an existing piece of *module*-based IP will not work. It would be an interesting enhancement to the language to permit interfaces to instantiate *modules*. Alternatively, most of the examples in this paper could be *module*-based if that construct permitted hierarchical access to its members.

9. CONCLUSION

In traditional Verilog, there were limited ways to encapsulate and abstract portions of RTL code. While the traditional *module* provides encapsulation, its portlist makes for a clumsy and verbose connection with procedural code. With SystemVerilog, there is some hope with the *interface* construct. When used as “pseudo modules”, *interfaces* permit the designer to encapsulate common and pervasive code into predefined templates and use them as objects in design descriptions.

10. ACKNOWLEDGMENTS

Thanks to Neil Johnson of XtremeEDA and to David Wellings of Thales Systems Canada for their feedback and input.

11. REFERENCES

[1] IEEE P1800™/D6(colored) Draft Standard for SystemVerilog:Unified Hardware Design, Specification and Verification Language

[2] Object-Oriented Analysis and Design with Applications 2nd edition, Grady Booch, 1994

[3] Synthesizable Finite State Machine Design Techniques Using the New SystemVerilog 3.0 Enhancements, Cliff Cummings, SNUG 2003

[4] SystemVerilog in Use : First RTL Synthesis Experiences with Focus on Interfaces, Peter Jensen, Thomas Kruse, Wolfgang Ecker, SNUG Europe 2004

[5] SystemVerilog : Interface Based Design, Peter Jensen, Thomas Kruse, Martin Zambaldi

[6] Pragmatic Simulation-Based Verification of Clock Domain Crossing Signals and Jitter using SystemVerilog Assertions, Mark Litterick, DVCON 2006

[7] Wikipedia : http://en.wikipedia.org/wiki/Object-oriented_programming

[8] Functional Triple Modular Redundancy (FTMR) VHDL Design Methodology for Redundancy in Combinatorial and Sequential Logic, Sandi Habinc, Gaisler Research, 2002