

SystemVerilog: Interface Based Design

Peter Jensen
SyoSil Consulting
peter@syosil.dk

Wolfgang Ecker, Thomas Kruse,
Martin Zambaldi
Infineon Technologies
{wolfgang.ecker, thomas.kruse,
martin.zambaldi}@infineon.com

Abstract

After establishing of various possibilities of abstraction in HDLs (on values, time, and structure) many years ago, SystemVerilog as a combined HDVL offers a new approach to support also abstraction on ports. This interface concept extends the feasibilities for encapsulation when designing, connecting, and verifying the numerous interfaces in modern SoC designs. This can be done on an abstract, non-synthesizable level but also on RT level.

The introduction of interfaces into SystemVerilog allows a more effective methodology for the implementation and verification of designs. First, an interface can be designed independently on the sub-blocks, which should be connected later. It contains not only the protocols for the transfers but also assertions derived from the specification to check all these transfers. Then, BFMs can be implemented using the interfaces and can be verified against the interface assertions. After implementation of the RTL code, this RTL code can be verified by using the BFMs and the assertions. Later, sub-designs are connected hierarchically whereby the assertion and the BFMs are used as the lowest level of the testbench hierarchy.

SystemVerilog has a lot of benefits against traditional HDLs as VHDL or Verilog and also against HVLs, as it combines many well-known concepts in a pragmatic way.

1 Introduction

Most professionals working with design and verification of integrated hardware devices currently experience an increasing lack of efficiency in their daily working methods and the EDA tools they employ. This is caused by the constantly increasing “Design and Verification Gap,” which describes the difference between the possible capacity and complexity of integrated systems of today, and the productivity of the single design and verification engineer. Various tendencies try to close the gap. Reusable Intellectual Property (IP) blocks are widely employed, larger chip areas are used for simple structures such as memories, and engineers seek to find new methods for designing and verifying larger systems in an adequate time.

A brief glance of the design methods used today is not too encouraging. The single method for speeding up the design time is to reuse IP extensively. The EDA tools supporting RTL design have improved – synthesis tools have evolved to be “physically driven,” and the improved computing power enables the tools to grasp larger systems at once. Except for IP reuse, it is remarkable that the design abstraction used in the industry has failed to evolve. When creating new systems and modules, a designer writes the same RTL style as was done in the industry ten years ago, using languages such as Verilog’95 and VHDL’87, possibly using various pre-processors. The academic world and the EDA industry have made several attempts to enable logic synthesis of hardware description languages (HDLs) at a higher abstraction level than what the RTL offers. These attempts have only caused little or no industry impact.

More advancement has been achieved in the shift of verification methods. Simulation-based verification using self-checking tests with or without functional reference model comparison is still widely used, but a significant shift has happened towards assertion-based and coverage-driven verification. Such methods are based on exercising a design with constrained random stimuli, while trying to obtain full functional coverage of the design. Assertions are used to describe the expected

behavior of the design, this means, they act as a specification. Such assertions may be checked continuously during simulation, but can also be used for formal (static) verification, revealing whether the assertions always hold for all possible dynamic simulations.

Such new verification methods have led to the introduction of hardware verification languages (HVLs), for describing constraints for random generation, coverage goals, and assertions. Also, such languages support the making of test benches with embedded functional reference models. These HVLs are typically used in conjunction with special verification tools, which are chained to the simulator engine, potentially slowing down simulation speed. Furthermore, an HVL is in some cases only supported by a single EDA tool, causing the verification to be dependent on a single tool, which increases the EDA related costs. Verification IP written in an HVL might therefore not be portable across tools – and may not be sellable either.

The combined HDL and HVL SystemVerilog 3.1a [SV_LRM] addresses these concerns related to design and verification. Improved RTL modeling capabilities are included together with a full HVL functionality, while being backwards compatible with the Verilog'95 and Verilog'2001 standards. This enables the design and verification engineers to work using one single joint language, while being able to port complete design and verification systems from one EDA environment to another. When using SystemVerilog for design, an experienced RTL designer will find that the language lifts Verilog to the VHDL RTL level – and beyond. Complex data types, improved constructs for conditional procedural expressions and processes are included. Most remarkable is the SystemVerilog *interface* construct, which allows RTL modeling on a higher abstraction level.

This paper will show how to employ the *interface* to synthesize an abstract, generic, multiplexed bus subsystem. Complex usage of the *interface* for construction of bus functional models is shown along with how to embed SystemVerilog assertions into the *interface*.

2 Interface Basics

A significant amount of complexity encountered, while designing the integrated SoC's of today has shifted from creating design blocks towards reusing existing IP components and existing systems. Often such systems have well defined, standardized bus systems on-chip, to which the various IP blocks are connected. This design methodology calls for a shift in the languages used for design, as well as for verification.

A classic RTL design language has its prime focus upon inferring state holding elements and combinational logic. Less focus is put upon the interfaces between various design blocks, where especially Verilog enforces a very low abstraction level, resulting in verbose and error prone typing, when binding blocks together.

With the *interface*, SystemVerilog offers a construct that introduces a higher level of abstraction and extends the methods for encapsulation when designing, connecting, and verifying the numerous interfaces in modern SoC designs. This applies both to the RTL layer and more abstract non-synthesizable verification components.

A SystemVerilog *interface* is a single unit capable of encapsulating a complete interface with elements such as signals, state holding elements, control logic, and the verification components characterizing the allowed behavior of the interface. The use of the *interface* construct forces design teams to adhere to precisely defined interfaces between design blocks. Also, a new dimension is added to the conventional top-down design methodology, as blocks can be tied together with an abstract interface at a very early point in the design process, before the final wiring is known. The *interface* construct enhances the ability to work with systems enclosing blocks being at various abstraction levels, as well as it allows blocks to connect to generic bus systems, where access to the interface is done by invoking abstract methods defined within the interface.

SystemVerilog assertions and verification components can be embedded into the *interface* construct. These assertions can be used to completely characterize the set of valid transactions on the interface, and thus enable continuous checking while performing simulation-based and coverage-driven verification. Further, such assertions allow formal methods to be invoked to prove that the interface clients fulfill certain restrictions imposed by the interface.

Such embedded interface verification elements greatly increase the efficiency of finding the functional errors in the attached interface clients. Further, it allows a vendor of bus standards to package the verification components with the system and RTL models delivered to the end user.

1 Interface Elements

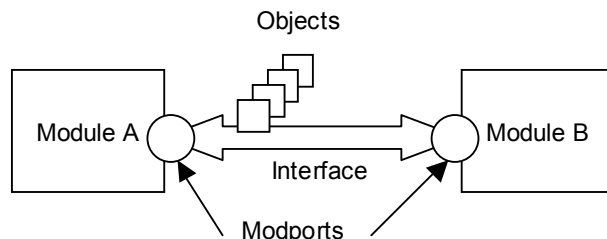
The SystemVerilog *interface* comprises the following elements.

- Objects (data carriers in the form of variables and/or wires)
- Methods (functions/tasks for handling the objects in the interface)
- Processes (static present functionality within the interface, e.g. *always* statements)
- *modports* (connectivity and accessibility definitions)

While the objects and processes are elements existing within the interface, interface methods are offered to the connected clients for accessing, handling, and manipulating the interface objects. Interface processes can also invoke the interface methods.

The *modport* construct is used to define accessibility between a client and the interface. If objects are listed, a client connected using that *modport* is enabled to access named objects, possibly using one of several access modes (types *input*, *output*, or *inout*). Access to interface methods can also be controlled. *modports* can be omitted when designing non-RTL interfaces, however they are mandatory for RTL synthesis.

In the most basic form, an interface can be viewed as a bundle of wires (objects), connecting a number of module instances (clients), as shown in the figure below.



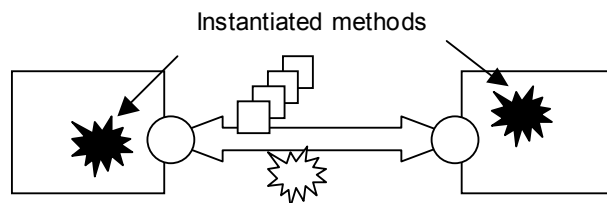
Basic Interface

```
interface bus;
bit req, ack;    // objects

modport mp_a (   // modport
input req,
output ack);
modport mp_b (   // modport
output req,
input ack);

endinterface
```

To preserve the abstractness of the interfaces, clients preferably should access the interface objects using methods defined within the interface. The figure below shows how the clients import methods (SystemVerilog functions and tasks) from the interface. The language also supports export of methods defined by the clients to the interface.



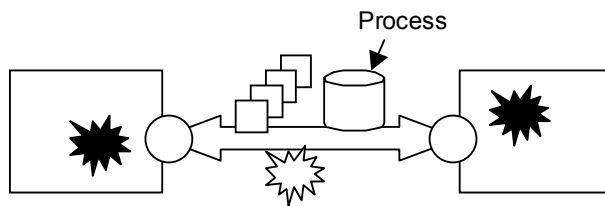
Interface with Methods

```
// interface defined methods
function send (...); ...
function get (...); ...

// clients import methods
modport mp_a (
input req, output ack,
import function send());
modport mp_b (
output req, input ack,
import function get());
```

More complex interface types might include processes, such as bus controllers and arbiters, performing required interface maintenance tasks. Processes within interfaces are written similar to ordinary RTL code – *always* blocks and/or continuous *assign* statements accessing the objects within

the interface. The presence of processes is not directly visible to the interface clients, but is powerful as they allow moving of interface/bus specific functionality from the clients to within the interface.



```

bit req1, req2;
bit [15:0] addr;
// intface process
always_comb begin
  if (addr[15]) begin
    req1 = req;
    req2 = 1'b0;
  end else begin
    req1 = 1'b0;
    req2 = req;
  end
end
end

```

Interface with Processes

2 Interface Instantiation and Connection

The connection of a client to an interface can be done using either a generic connection or a non-generic connection. The generic style allows a client to be developed without knowing the name of the interface, or the object names within the interface. Rather, the client will use interface methods to access the interface. The non-generic style is based on developing clients, which only connects to a named interface, while accessing the interface objects directly.

Use of the generic interface approach should be encouraged in order to enhance the level of abstractness and reusability of the interfaces and clients. With this style, the client can be completely or partly independent on the interface/bus protocol.

```

module client_gi (
  interface i; // generic intface
);
  always_comb begin
    if i.send(...) ...
  end
endmodule

module toplevel ();
  bus bus_i (); // intf. instant.

  client_gi i1 ( // client instant.
    .i(bus_i.mp_1) // using modport
  );
endmodule

```

Generic Interface
Client uses interface methods only

```

module client_ngi (
  bus i; // non-generic intf.
);
  always_comb begin
    if (i.ack) i.req = ...
  end
endmodule

module toplevel ();
  bus bus_i (); // intf. instant.

  client_ngi i2 ( // client instant.
    .i(bus_i.mp_2) // using modport
  );
endmodule

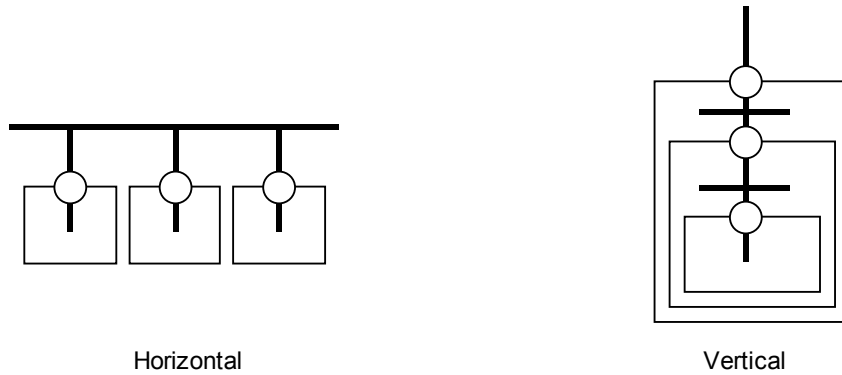
```

Non-generic Interface
Client uses interface objects directly

3 Interface Topology

An interface connecting multiple modules can topologically be characterized in two different categories. A horizontal interface connects multiple top-level modules, resembling a bus system. A vertical interface connects the various hierarchical levels of a single module.

The diagram below outlines these two topologies, and displays where *modports* are utilized. One should note that these two topologies can and will be used in a mixed style.



Interface Topologies

The horizontal interface is well known and described, including in this paper.

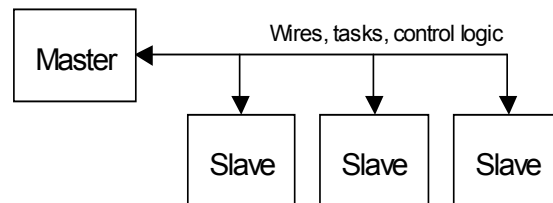
The vertical interface is rare, and exposes some problems related to the SystemVerilog *modports* and their use. Each *modport* used throughout the hierarchies of the vertical interface design must be described in the interface declaration. This implies that the interface declaration must define what hierarchical level in the design drives a certain interface object. This contradicts normal top-down and module-based design approaches, where a local module on top level is unaware of where signal drivers are located in the hierarchies below. We would like to suggest that *modports* could be added locally to an interface to resolve this language deficiency, and that the SystemVerilog LRM explicitly describes the use of vertical interfaces.

Generic interfaces do not mix very well with vertical interfaces, as each hierarchical crossing require the use of a named *modport* from the generic interface. Typically, a generic interface client does not know such information.

We have also experienced that EDA tools interprets a *modport* as a separate name space (view) inside the interface. This implies that a client only can see the *modport* definition used to connect the client to the interface. As a consequence, a vertical interface configuration must use same *modport* for every hierarchical crossing in the design. This allows for an interface to be connected across several vertical hierarchies, but the interface must be dissolved into a group of objects before being distributed to and written by multiple hierarchies. This inhibits the use of methods imported from the interface.

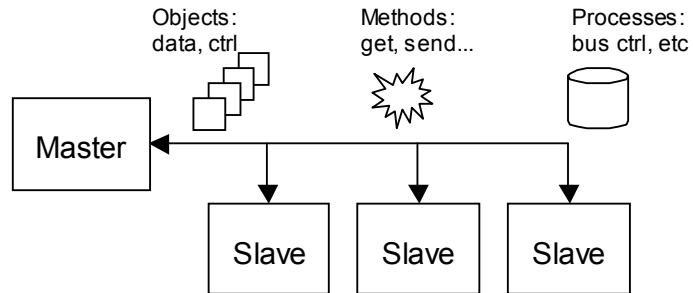
4 Top-Down RTL Refinement of Bus System Behavioral Model

In this section we will show how to migrate an interface encapsulated abstract behavioral bus subsystem using a top-down approach. The target of this migration is to refine and model the bus protocol in RTL while keeping the protocol-specific parts within the interface, and allow for the bus clients to access the bus without knowing the protocol. Clients built using this approach can be reused and easily attached to other systems with a different bus protocol, assuming that such other systems offer same method-based API-like interface. Consider following trivial horizontal bus subsystem:



Trivial Bus Subsystem

If modeled at all, a conventional behavioral Verilog'95 model of this subsystem would be a number of wires connecting the master with the slaves. Furthermore, one or more processes would make up the bus control, and tasks/functions could be available for the master and slaves to handle bus transactions. Important is that the detailed wiring already must be determined in a behavioral model. In many real-world applications, the bus protocols will be hard-coded in the masters and slaves. Putting the example into a SystemVerilog perspective and using the interface construct does not alter the block diagram much:



Bus Subsystem using Horizontal Interface

The big difference is that no specific wiring is connected to the master and the slaves. Rather, an abstract interface is connected – the modules do not even need to know the name of the interface type. The interface clients now access the interface using abstract methods supplied by the interface. Below, partial pieces of the relevant SystemVerilog behavioral code are shown.

```
interface bus ();
    parameter int no_slaves = 0;
    dataword ma_data;
    addrword ma_addr;
    always begin
        // bus control process
        // direct data to addressed slave
    end
    task automatic ma_send (...); ...
    task automatic sl_receive (...); ...
    function bit sl_datardy (...); ...
endinterface

module toplevel ();
    bus #(no_slaves(3)) bus_inst ();
    master m (.b(bus_inst), .*);
    slavel #(.slave_id(0)) s1 (.b(bus_inst), .*);
    slave2 #(.slave_id(1)) s2 (.b(bus_inst), .*);
    slave3 #(.slave_id(2)) s3 (.b(bus_inst), .*);
endmodule
```

SystemVerilog Behavioral Model of Bus Subsystem
Interface Declaration and Instantiation

```

module master (
    interface b,
    // other master specific io's
);
    dataword data;
    always begin
        // master specific code
        b.ma_send(data);
        // more code
    end
endmodule

module slave1 (          // slave2 / slave3 built similarly
    interface b,
    // other slave1 specific io's
);
    parameter int slave_id = 0;
    dataword data;
    always begin
        // slave specific code
        if (b.sl_datardy(slave_id))
            data = b.sl_receive(slave_id);
        // more code
    end
endmodule

```

SystemVerilog Behavioral Model of Bus Subsystem Usage of Interface

When refining the behavioral model towards RTL, the limiting factor becomes the synthesis tool. Tasks used in behavioral modeling presumably contains a number of wait statements, for instance for rising clock edges. Typically, most synthesis tools do not support synthesis of such tasks, as they implicitly would require the inference of finite state machines.

Below, a methodology for RTL modeling of interface based bus systems is proposed, which

- retains RTL code required to handle the bus protocol within the SystemVerilog interface
- supports multiplexed buses
- supports a single master and multiple different slaves

When handling bus protocols of just minor complexity, the interface client requires a finite state machine to handle the protocol. The client then knows in what states data must be sent and received, and how to operate the control signals on the bus.

Our prime goal is to implement a slave with little or no knowledge about the bus protocol, but still to have the bus control logic inferred within the slave. This would allow for a synthesized IP component to be attached to a bus system with the same protocol as the one dictated by the SystemVerilog interface.

The RTL code within the slave should only concentrate on sending and/or receiving data and implementing the custom function of the slave. In order to obtain this, we want to retrieve protocol specific information from the SystemVerilog interface, packaged as functions and tasks. The methods declared in the interface are instantiated together to form an FSM. One method infers the state holding elements, while the other method infers the logic driving the FSM outputs. Other methods declared in the interface are invoked to access the bus (e.g., get or send data). These methods are all without state holding elements.

The SystemVerilog RTL code below implements a simple example of such a bus system, derived from the behavioral model shown previously.

```

interface bus ();
  parameter int no_slaves = 0;
  dataword ma_data;
  addrword ma_addr;
  bit ma_req;
  bit [no_slaves-1:0] sl_ack, sl_req;

  always_comb begin // bus control processes
    // Set sl_req[ma_addr] = ma_req
    // Set ma_ack = sl_ack[ma_addr]
  end

  // functions for ma/sl interf. access
  function bit ma_send (...); ...
  function dataword sl_receive (...); ...
  function bit sl_datardy (...); ...

  // data types and data carriers for FSMs
  typedef enum int {sl_rdy, sl_busy, ...} sl_statetype;
  typedef enum int {ma_rdy, ma_busy, ...} ma_statetype;
  ma_statetype ma_state;
  sl_statetype [no_slaves-1:0] sl_state;

  // methods for FSM inference in ma/sl
  task automatic sl_fsm_nxt(); ... state <= ...; endtask
  function void sl_fsm_comb(...); ...
  task automatic ma_fsm_nxt(); ... ma_state <= ...; endtask
  function void ma_fsm_comb(); ...

  // modports
  modport master (
    output ma_data, output ma_addr, output ma_req, output ma_state,
    input ma_ack,
    import function ma_send(), import task ma_fsm_nxt(),
    import function ma_fsm_comb() );
  modport slavel (
    input ma_data, input sl_req,
    output .ack(sl_ack[0]), output .state(sl_state[0]),
    import function sl_receive(),
    import function sl_datardy(),
    import task sl_fsm_nxt(),
    import function sl_fsm_comb() );
  // further modports for slaveN
endinterface

module toplevel ();
  bus #(no_slaves(3)) bus_inst ();
  master m (.b(bus_inst.master), .*);
  slavel #(.slave_id(0)) s1 (.b(bus_inst.slave1), .*);
  slavel #(.slave_id(1)) s2 (.b(bus_inst.slave2), .*);
  slavel #(.slave_id(2)) s3 (.b(bus_inst.slave3), .*);
endmodule

```

SystemVerilog RTL Model of Bus Subsystem Interface Declaration and Instantiation

```

module master (
    interface b, // generic interface
    // other master specific io's
);
    // Infer FSM based on interf. inform.
    always_ff @(posedge clk) b.ma_fsm_nxt();
    always_comb b.ma_fsm_comb();

    dataword data;
    bit datardy, bit ack;
    // when data is available send to sl.
    // check if acknowledge
    always_comb ack = b.ma_send(data, datardy);

    // more master specific code, create 'data'
endmodule

module slavel ( // slave 2 / 3 built similarly
    interface b, // generic interface
    // other slavel specific io's
);
    parameter int slave_id = 0;

    // Infer FSM based on interf. inform.
    always_ff @(posedge clk) b.sl_fsm_nxt(state, slave_id);
    always_comb b.sl_fsm_comb(ack, slave_id);

    dataword data;
    // when data is ready get from intf.
    always_ff @(posedge clk)
        if (b.sl_datardy(slave_id)) data <= b.sl_receive(slave_id);

    // more slave specific code, handle 'data'
endmodule

```

SystemVerilog RTL Model of Bus Subsystem Usage of Interface

This example employs the SystemVerilog3.1a *modport expression* to allow slavel to access the objects *sl_ack[0]* and *sl_state[0]* by the names *ack* and *state*. The use of this construct is necessary to allow multiple interface clients to write to single interface objects.

Unfortunately, using the *modport expression* feature inhibits direct access to interface objects from methods defined within the interface, as the object renaming is not visible in the scope of the method. This requires the methods *sl_fsm_nxt* and *sl_fsm_comb* to access *state* and *ack* via the function parameter list, and these objects must therefore be explicitly named within the slave interface client(s), which is a drawback. We are therefore currently encouraging the SystemVerilog committees to enhance the *modport expression* feature.

3 Interface Synthesis

The SystemVerilog LRM does not explicitly define what language subset can be synthesized. This is a drawback, as a standardized synthesizable subset of the language will make it easier for users to write code, which can be ported between different EDA vendor tools.

One of the most widespread RTL synthesis tools Design Compiler [Synopsys] has support for a significant subset of SystemVerilog. Based on the experiences with that tool, we here summarize what subset of the *interface* construct is synthesizable using the available EDA technology of today.

Synthesizable	Not synthesizable
<i>interface</i>	
<i>interface</i> , generic	
<i>interface</i> objects, methods, processes	<i>interface</i> methods with multiple wait statements (tasks).
<i>interface modport</i>	<i>interface modport expression</i>
<i>interface modport</i> method <i>import</i>	<i>interface modport</i> method <i>export</i>
	nested combinations of <i>modules</i> and <i>interfaces</i>

If an interface is connected to a client without the use of a *modport*, all objects and methods are accessible. This might be quite useful when modeling using non-RTL code, but the absence of the direction type for the objects causes Design Compiler to set this to *inout* (i.e., similar to conventional tri-state type wires). In modern RTL designs such port types are unwanted and prohibited to use. Therefore to write a synthesizable model, *modports* are mandatory when connecting each interface client.

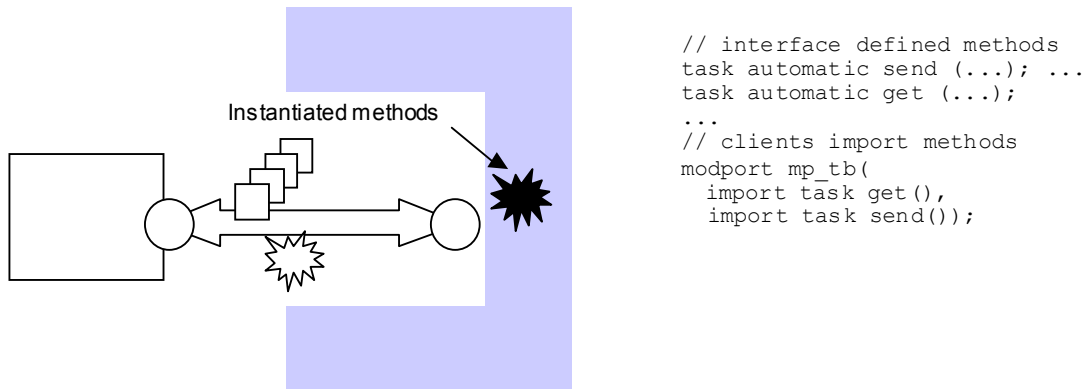
The extensive RTL example in the above section can be synthesized using synthesis technology of today, except for the use of the employed *modport expression* feature. The *modport expression* should be made synthesizable as well, to enable the construction of parameterizable interface designs.

SystemVerilog opens up for wider use of behavioral, abstract interface modeling as shown in the above example. We believe that synthesis tools should evolve to enable synthesis of interface methods with multiple wait statements, which would imply inference of finite state machines directly from behavioral models.

4 BFM's using Non-Synthesizable Methods

Methods (i.e., functions and tasks) can be associated with interfaces. It was shown, that they can be used to make synthesizable models more abstract. Because synthesis tools limit their application, their maximum benefit cannot be achieved in conjunction with RT-synthesis.

However, if a *Bus Functional Model* is modeled, which needs not be synthesizable, methods can be used to model cycle-true transactions (i.e., waveforms and expected waveforms on a bundle of signals). This application is shown in the subsequent figure.

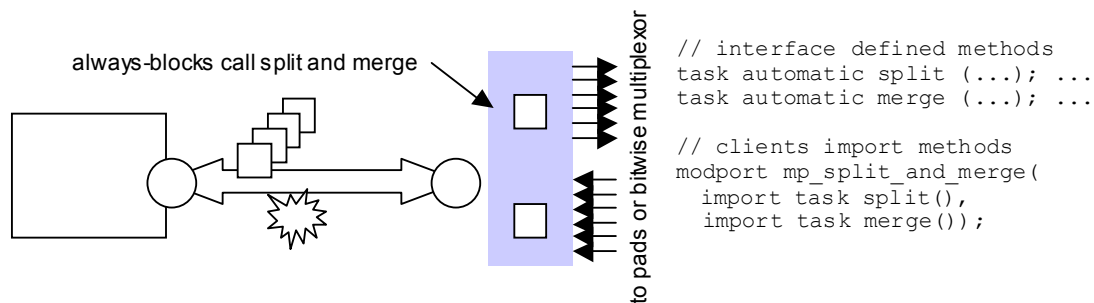


BFM with testbench interfaces

First, the methods *send* and *get* are declared. The procedural statements *@(posedge ...)* or *@(negedge ...)* are used to synchronize the actions in the method with the clock, and are thus responsible for time advance in the method. Setting, checking, and reacting on values at specific times and for specific signals are used to model the signal values of the waveforms. The BFM methods can be included in other methods to provide more complex BFMs. This is exemplarily done in a method *send_stream*, which iteratively calls the *send* to force the transmission of a complete data package. If the methods does not contain only sequential but also concurrent actions (e.g., using *fork-join* constructs), pipelined transactions can be composed from sequential transactions and encapsulated in methods.

All BFM methods are named in the *modport mp_bfms*. When the interface of the test bench is connected to this *modport*, all BFM methods can be called from the test bench. In this way, the writing of the test bench need not start from scratch and need not know every detail of the protocol of the interface.

Another application of methods is splitting and merging of variables. Their application is generally shown in Figure “Splitter and Merger” subsequently.



BFM with split and merge

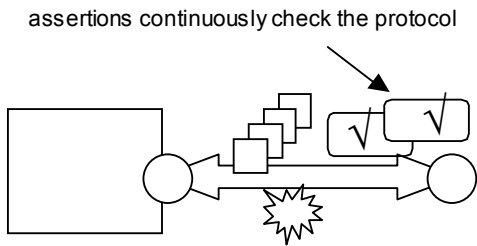
Splitter and merger methods are needed to connect the bundled signals of an interface to single signals. This application is needed for instance to connect a test bench intended for RT and cycle-true models to a corresponding gate-level model. Splitting up signals is not only required to connect the interface to single signals but also to delay single signals separately or to check the delay of single signals separately. To do so, the methods *split* and *merge*, which are made visible via a separate interface, are called in two *always*-blocks to extract and merge the signals. Not shown is the possibility to encapsulate the *always*-block in separate modules.

Also abstract, time-less interface methods can be used to model abstract communication. To transfer data at a glance and to provide general synchronization, a variable for transferred data and two variables of type *event* are needed. This means the concrete interface must be extended with variables. Also specific methods, here *abstract_send* and *abstract_get*, are needed.

The benefit of the additional abstract methods is that a structural model of the design can be composed generally. With configurations, either non-timed behavioral models, timed behavioral models, or RT-models can be realized.

5 BFMs with Embedded Checks using SystemVerilog Assertions

To validate the correctness of protocols of an interface or even better, to define the intended behavior of a protocol upfront (see the later Section Methodology), assertions can be included in interfaces. This is generally shown in the next picture.

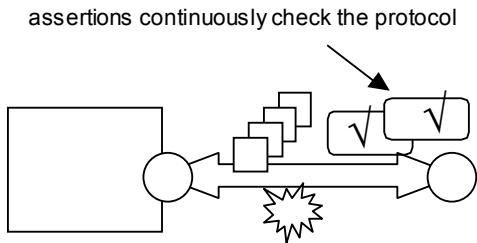


```
// Assertions
assert property
  @posedge(clk)
  disable iff (!reset)
  (rose(a)) |-> b;
assert property
  @posedge(clk)
  disable iff (!reset)
  (b) => (!b);
```

Assertions associated with interfaces

Assertions can be used to check correctness of the protocol at a glance but also to check aspects of the protocol such as response of acknowledge, the lack or existence of pipelining, just to name two examples.

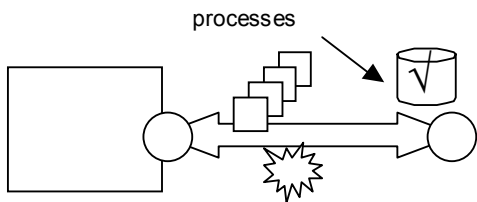
Further on, assertions can also be used to collect coverage information, which is later used to evaluate the completeness of the test cases. To do so, it is recommended to encapsulate the protocol checker in a property and then either check or cover that property. Control over functionality can be gained either by encapsulating the checker in generate statements, in *ifdef* clauses, or by using the SystemVerilog system calls for assertion control. Such an application is shown next.



```
// Assertions
property p1 is
  @posedge(clk)
  disable iff (!reset)
  (rose(a)) |-> b;
generate
  if (ASSERT_P1)
    assert property p1;
  end if;
end generate;
```

Controlled assertions associated with interfaces

Especially for sequential and erroneous pipelined protocols, assertions can also be modeled using always blocks. Their description style is similar to BFM protocol receivers and data extractors. Immediate assertions are used to check values and to report violations. This is shown in the next figure.



```
// sequential checkers
always @posedge(clk)
  if (a && b)
    assert c;
```

Assertions as protocol parsers and sequential checkers

Always blocks are easier to use for designers familiar with sequential code. In addition, always blocks can be used to describe more complex checks than concurrent assertions alone.

Vice versa, protocol checks and protocol parsers could be modeled using assertions. Here, values are passed to variables outside the assertion by using statements, which are executed in case of pass or fail of the assertion. However, it should be mentioned that protocol generators and analyzers are implemented differently from protocol checkers.

6 Conclusion

After establishing of various possibilities of abstraction in HDLs (on values, time, and structure) many years ago, SystemVerilog as a combined HDVL offers a new approach to support also abstraction on ports. This interface concept extends the feasibilities for encapsulation when designing, connecting, and verifying the numerous interfaces in modern SoC designs. This can be done on an abstract, non-synthesizable level but also on RT level.

The introduction of interfaces into SystemVerilog allows a more effective methodology for the implementation and verification of designs. First, an interface can be designed independently on the sub-blocks, which should be connected later. It contains not only the protocols for the transfers but also assertions derived from the specification to check all these transfers. Then, BFM's can be implemented using the interfaces and can be verified against the interface assertions. After implementation of the RTL code, this RTL code can be verified by using the BFM's and the assertions. Later, sub-designs are connected hierarchically whereby the assertion and the BFM's are used as the lowest level of the testbench hierarchy.

On the other hand, we see in SystemVerilog especially in its interface concept several aspects, which should be improved to make the language more easily usable even for complex designs. So, a concept is missing to group procedure blocks (e.g., always blocks) inside of interfaces. It is not allowed to use modules (neither declaration nor instantiation) here. There is also no concept to use interfaces in a traditional top-down approach. To define an interface on top-level, information about the lower levels are needed already. A possible solution would be the support of local modports or the extension of interfaces. A further issue is the insertion of clocking and reset domains. Because in general several interfaces are connected to one sub-design, the same clock or reset may also be connected multiple times to this sub-design without having a possibility to give this information to the sub-design. This could cause some problems especially on RT level.

But in spite of these weaknesses, we think that SystemVerilog has already today a lot of benefits against traditional HDLs as VHDL or Verilog and also against HVLs. SystemVerilog combines many well-known concepts in a pragmatic way.

Further on, we claim that only SystemVerilog standardization under Accelera as well as under IEEE and the availability of EDA tools gives the chance to apply SystemVerilog to real designs. In this way, language enhancements needed for design and verification can be gathered and feed back into the SystemVerilog language development..

7 References

- [SV_LRM] Accellera: SystemVerilog 3.1a Language Reference Manual, www.systemverilog.org

- [Synopsys] Synopsys: "SystemVerilog Synthesis User Guide", version V-2003.12, December 2003

- [Pieper04] Pieper, Karen L.: "How SystemVerilog aids design and synthesis", www.eedesign.com, Jan 22, 2004

- [Jensen04] Jensen, Kruse, Ecker: "SystemVerilog in Use: First RTL Synthesis Experiences with Focus on Interfaces", SNUG Europe, May 2004