SystemVerilog 3.1
C/C++ API Tutorial

João Geada

Michael Rohleder

Doug Warmke

Outline

- Why does SystemVerilog (SV) need an enhanced API
- How the standard was developed
- The enhanced SV 3.1 APIs
    - Direct Programming Interface (DPI)
    - Consistent method for loading foreign code
    - VPI extensions for Assertions
    - VPI extensions for Coverage
- How it all comes together: packet router example
- Open issues and further plans

Disclaimer:

This is a discussion of work under development.

The exact details of syntax and usage described by this presentation are still under discussion.

Some changes to the material shown might be experienced in the final form of SystemVerilog.

Why does SV3.1 need new APIs

- VPI and PLI are not easy interfaces to use
  - Require knowledge of the interface for even trivial usage
  - A significant percentage of users do not need the sophisticated capabilities provided by VPI/PLI. They only need a way of invoking foreign functions from SV and getting results back
- VPI and PLI are not symmetric: Verilog can invoke C functions but C functions cannot invoke Verilog functions
- SV has new data types, including user defined structs and unions, that are not part of the VPI object model
- SystemVerilog includes assertions. These are a significant addition to the language and were not addressed by any prior Verilog API
- Coverage driven tests have become a well established practice, but no standard mechanism was available to implement such testbenches

How the standard was developed

- DPI and the VPI extensions are based on production proven donations from Synopsys
  - DirectC interface
  - Assertions
  - Coverage
- The SV-CC committee accepted those donations and integrated them into the framework of the SV language
- Foreign code loading mechanism proposed by Motorola

SystemVerilog-C/C++ Comittee

- Representatives of user and vendor companies
- All major EDA companies are represented

John Amouroux, Mentor
Kevin Cameron, National
João Geada, Synopsys
Ghassan Khoory, Synopsys, Co-Chair
Andrzej Litwiniuk, Synopsys
Francoise Martinole, Cadence
Swapanjit Mittra, SGI, Chair
Michael Rohleder, Motorola
John Stickley, Mentor
Bassam Tabbara, Novas
Doug Warmke, Mentor
Simon Davidmann, Synopsys
Joe Daniels, LRM Editor
Peter Flake, Synopsys
Emerald Holzwarth, Mentor
Tayung Liu, Novas
Michael McNamara, Verisity
Darryl Parham, Sun
Tarak Parikh, @HDL
Alain Reynaud, Tensilica
Stuart Swan, Cadence
Kurt Takara, 0-in
Yatin Trivedi, ASIC Group, Past Chair

see URL: www.eda.org/sv-cc

DPI - Overview

- Name DPI = Direct Programming Interface (work name)
- Natural inter-language function call interface between SV and a foreign programming language (e.g. C)
- Relies on C function call conventions and semantics
- On each side, the calls look and behave the same as the normal function calls for that language
  - On SV side, DPI calls look and behave like any other SV function
  - On C side, DPI calls look and behave like any other C function
- Binary or source code compatible
  - Binary compatible in absence of packed data types (svc.h)
  - Source code compatible otherwise (svc_src.h)

DPI - Declaration Syntax

- Extern functions (implemented in C, called from SV):

  extern "DPI" <attrprop> <result> [cname= <name>] (<params>);

  - name shall not start with $, must adhere to C function naming
  - param = [<direction>] <type> [<param_name>]; input direction is default
  - <result> identifies return type, void functions return no value
  - cname is optional, defaults to fname.
    cname, if given, is the name of the function as seen by C.
- Export functions (implemented in SV, called from C):

  export "DPI" [cname= <name>];
- Extern declaration is in same scope as function call site
- Export declaration is in same scope as function definition

DPI - Basics

- Formal arguments: input, inout, output + return value
  - input arguments shall use a const qualifier on the C side
  - output arguments are uninitialized
  - passed by value or reference, dependent on direction and type
- Shall contain no timing control; complete instantly and consume zero simulation time
- Changes to function arguments become effective when simulation control returns to SV side
- Memory ownership: Each side is responsible for its allocated memory
- Use of **ref** keyword in actual arguments is not allowed

DPI – Function Properties/Attributes

- Possible Function Properties/Attributes are:
  - *pure:* no side effects/internal state (I/O, global variables, PLI/VPI calls) result depends solely on inputs, might be removed when optimizing
  - *context:* mandatory when PLI/VPI calls are used within the function, or when an imported C/C++ function calls an exported SV function
  - *(default):* no PLI/VPI calls, but might have side effects
- Free functions have no relation to instance specific data
- Context functions are bound to a particular instance
  - Can work with data specific to that module / interface instance
  - For C calling SV, it is expected that context functions will predominate
  - Context aware functions are useful for implementation of functions attached to specific instances

DPI – Parameter Passing

- Most SV data types are supported
- Value passing requires matching type definitions on both sides
  - user is responsible to ensure this
  - packed types: arrays (defined), structures, unions
  - arrays see next slide
- Function result types are restricted to small values and packed bit arrays up to 32 bits and all equivalent types
- Usage of packed types might prohibit binary compatibility

SV type	C type
char	char
byte	char
shortint	short int
int	int
longint	long long
real	double
shortreal	float
handle	void*
string	char*
bit	(abstract)
enum [<type>]	<type>|int
logic	avalue/bvalue
packed array	(abstract)
unpacked array	(abstract)

DPI – Parameter Passing (Arrays)

- Array parameters are passed by handle of type svHandle
- Arrays use normalized ranges for the packed [n-1:0] and the unpacked part [0:n-1]
- A formal argument name must separate the packed and the unpacked dimensions of an array
- Open Array have an unspecified range for at least one dimension
    - Relaxation of argument matching rules, range is ignored for a match
    - Denoted by [] in the function declaration; elements can be accessed in C by the same range as defined in SV for the actual argument
    - Query functions are provided to determine array information
    - Library functions are provided for accessing the array
- Examples:
  bit [15:8], logic [31:0], logic [] 1x3 [3:1], bit [] unsized_array []

DPI – Argument Coercion (extern)

- For **external** functions, arguments appear in 3 places:
  - Formal arguments in C-side function definition
  - Formal arguments in SV-side external function declaration
  - Actual arguments at SV-side call site
- There is **no coercion at all** between SV formals in an external function declaration and the corresponding C-side formals
- Normal SV coercions are performed between SV actuals at a call site and SV formals in the external declaration
- User is responsible for providing C-side formal arguments that precisely match the SV-side formals in the external function declaration

DPI – Argument Coercion (export)

- For exported functions, arguments appear in 4 places:
  - SV-side function definition
  - C-side function prototype
  - C-side function call site
- The SV simulator will **not perform argument coercion** for the C-calls-SV direction
- The programmer must provide C-side arguments that exactly match the type, width, and directionality requirements of SV formals

DPI – C and SV Argument Matching

- DPI's argument matching rules are based on common sense and they are straightforward. However, there are many details for users to remember
- To help users easily create DPI inter-language function calls, it is expected that most SV implementations will automatically generate C prototypes for each extern and export function encountered in the SV source deck

Consistent loading of foreign code

- Only applies to DPI functions, PLI/VPI not handled (yet)
- All functions must be provided within a **shared library**
  - user is responsible for compilation and linking of this library
  - SV application is responsible for loading and integration of this library
- Libraries can be specified by switch or in a bootstrap file
  - -sv_lib <filename w/o ext>
  - -sv_liblist <bootstrap>
  - extension is OS dependent; to be determined by the SV application
- Uses relative pathnames
  - -sv_root defines prefix

```
#!SV_LIBRARIES
# Bootstrap file containing names
# of libraries to be included
 function_set1
 common/clib2
 myclib
```

VPI extensions for Assertions

- Permits 3rd party assertion debugger applications
  - Usable across all SV implementations
- Permits users to develop custom assertion control mechanisms
- Permits users to craft C applications that respond to assertions
- Permits users to create customized assertion reporting mechanisms different than those that may be built into the used SystemVerilog tool

accellera

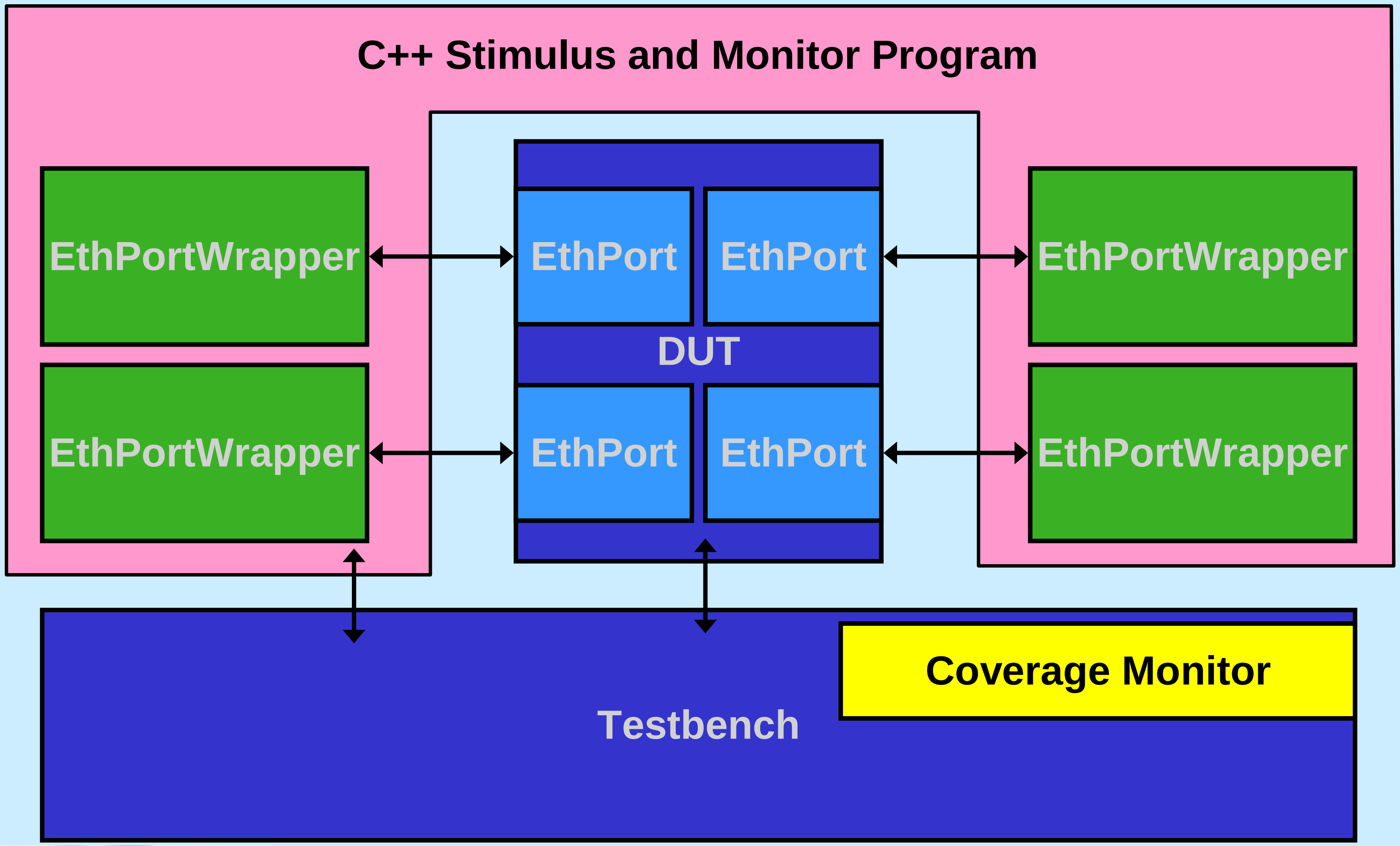
VPI for assertions: overview

- Iterate over all assertions in an instance or the design
- Put callbacks on an assertion
  - Success
  - Failure
  - Step
- Obtain information about an assertion
  - Location in source code where the assertion is defined
  - Signals/expressions referenced
  - Clocking signal/expression
  - Assertion name and directive and related instance, module
- Control assertions
  - Reset: discard all current attempts, leave assertion enabled
  - Disable: stop any new attempts from starting
  - Enable: restart a stopped assertion
  - EnableStep, DisableStep: enables/disable assertion stepping callbacks for an attempt

Coverage Extensions: SV and VPI

- Standardized definition for a number of coverage types
  - Statement, toggle, FSM state and assertion coverage defined
  - For these coverages, coverage data has same semantics across all implementations
- Defines 5 system tasks to control coverage and to obtain “realtime” coverage data from the simulator
  - $coverage_control, $coverage_get_max, $coverage_get, $coverage_merge, $coverage_save
  - Interface designed to be extensible to future coverage metrics without perturbing existing usage
  - Coverage controls permit coverage to be started, stopped or queried for a specific metric in a specific hierarchy of the design
- VPI extensions for coverage provide same capabilities as the system tasks above, plus additional “fine-grain” coverage query
  - Coverage can be obtained from a statement handle, FSM handle, FSM state handle, signal handle, assertion handle

Ethernet Packet Router Example



Example developed by John Stickley

C++ Side: SystemC Testbench Root

```
SC_MODULE(TestBench) {
    private:
        EthPortWrapper* context1;
        EthPortWrapper* context2;
        EthPortWrapper* context3;
        EthPortWrapper* context4;
        int numOutputs;

        void testThread(); // Main test driver thread.
    public:
        SC_CTOR(System) : numOutputs(0) {

            SC_THREAD(testThread);
            sensitive << UTick;

            // Construct 4 instances of reusable EthPortWrapper
            // class for each of 4 different HDL module instances.
            context1 = new EthPortWrapper("c1"); context1->Bind("top.u1", this);
            context2 = new EthPortWrapper("c2"); context2->Bind("top.u2", this);
            context3 = new EthPortWrapper("c3"); context3->Bind("top.u3", this);
            context4 = new EthPortWrapper("c4"); context4->Bind("top.u4", this);
        }
        void BumpNumOutputs() { numOutputs++; }
};

void TestBench::testThread() {
    // Now run a test that sends random packets to each input port.
    context1->PutPacket(generateRandomPayload());
    context2->PutPacket(generateRandomPayload());
    context3->PutPacket(generateRandomPayload());
    context4->PutPacket(generateRandomPayload());

    while (numOutputs < 4) // Wait until all 4 packets have been received.
        sc_wait();
}
```

C++ side: SystemC EthPortWrapper

```
#include “svc.h”

SC_MODULE(EthPortWrapper) {
    private:
        svHandle svContext;
        sc_module* myParent;

    public:
        SC_CTOR(EthPortWrapper) : svContext(0), myParent(0) { }
        void Bind(const char* hdlPath, sc_module* parent);
        void PutPacket(vec32* packet);

    friend void HandleOutputPacket(svHandle context, int portID, vec32* payload);
};

void EthPortWrapper::Bind(const char* svInstancePath, sc_module* parent) {
    myParent = parent;
    svContext = svHandleByName(svInstancePath);
    svSetUserContext(svContext, (void*)this);
}

void EthPortWrapper::PutPacket(vec32* packet) {
    PutPacket(svContext, packet); // Call SV function.
}

void HandleOutputPacket(int portID, vec32* payload) {
    svHandle svContext = svGetFunctionContext();
    EthPortWrapper* me = (EthPortWrapper*)svGetUserContext(svContext); // Cast -> C context
    me->myParent->BumpNumOutputs(); // Let top level know another packet received.

    printf("Received output on port on port %\n", portID);
    me->DumpPayload(payload);
}

```

SV side: SystemVerilog EthPort Module

```

module EthPort(
    input [7:0] MiiOutData,
    input MiiOutEnable,
    input MiiOutError,
    input clk, reset,
    output bit [7:0] MiiInData,
    output bit MiiInEnable,
    output bit MiiInError);

    extern "DPI" context void HandleOutputPacket(
        input integer portID,
        input bit [1439:0] payload);

    export "DPI" PutPacket;

    bit packetReceivedFlag;
    bit [1499:0] packetData;

    //
    // This exported function is called by the C side
    // to send packets into the simulation.
    //
    function void PutPacket(input bit [1499:0] packet)
        packetData = packet;
        packetReceivedFlag = 1;
    endfunction
 endmodule
```

SV side: SystemVerilog EthPort, contd.

```
    always @(posedge clk) begin      // input packet FSM
        if (reset) begin
            ...
        end
        else begin
            if (instate == READY) begin
                if (packetReceived)
                    state <= PROCESS_INPUT_PACKET;
            end
            else if (instate == PROCESS_INPUT_PACKET) begin
                // Start processing input packet byte by byte ...
            end
        end
    end

    always @(posedge clk) begin     // output packet FSM
        if (reset) begin
            ...
        end
        else begin
            if (outstate == READY) begin
                if (MiiOutEnable)
                    state <= PROCESS_OUTPUT_PACKET;
            end
            else if (outstate == PROCESS_OUTPUT_PACKET) begin
                // Start assembling output packet byte by byte ...
                ...
                // Make call to C side to handle the assembled packet.
                HandleOutputPacket(myPortID, outPacketVector);
            end
        end
    end
endmodule
```

SV side: Coverage monitor

```
module coverage_monitor(input clk)
  integer cov = 0, new_cov = 0, no_improvement = 0;

  always @(posedge clk) begin
     // count clocks and trigger coverage monitor when appropriate
  end

  always @(sample_coverage) begin
    // get the current FSM state coverage in the DUT and all instances below
    new_cov = $coverage_get(`SV_COV_FSM_STATE,
                        `SV_HIER, “DUT”);
    if (new_cov <= cov) begin
           // no coverage improvement
           no_improvement++
           if (no_improvement == 3) $finish();
    end
    else begin
           // coverage still increasing. Good!
           cov = new_cov;
    end
  end
endmodule
```

Open Issues and Further Plans

- Extend VPI object model
  to support the complete SV type system
  - extend VPI to cover all new elements of SystemVerilog
- Additional callback functions
  to match enhanced scheduling semantics
- Further enhancements to loading/linking
  - inclusion of source code, uniform PLI/VPI registration
- Dealing with C++ specifics

- All driven by experiences and user requests/needs