Hardware/Software co-verification using Specman and SystemC with TLM ports

Horace Chan
PMC-Sierra Inc, 8555 Baxter Place,
Burnaby, BC, Canada, V5A 4V7
horace_chan@pmc-sierra.com

Brian Vandegriend
PMC-Sierra Inc, 8555 Baxter Place
Burnaby, BC, Canada, V5A 4V7
brian_vandegriend@pmc-sierra.com

Abstract—**In modern ASIC/SoC design, the hardware and software have to work seamlessly together to deliver the functions, requirements and performance of the embedded system. To accelerate time-to-market and to reduce overall development cost, it is crucial to co-verify the software code with the hardware design prior to tape-out. The software team can start developing and debugging their code with the actual hardware RTL code to shorten their overall development cycle. The hardware team can use the software code to identify performance bottlenecks and incorrect functional behaviors early in the development cycle which helps to reduce the risk of increasingly expensive device revisions.**

The current approach to co-verification is primarily running the software on the embedded processor inside the hardware design, either within the simulator or with ICE (in-circuit emulation). The disadvantage of this approach is slow debug turnaround time and the higher cost is procuring and supporting a dedicated emulation box or FPGA platform. In addition, the software is running in isolation relative to the testbench, hence it is often challenging and inconvenient to integrate the software with other verification IP in the testbench.

In this paper, we will present an alternate approach on how to integrate the software driver into the simulator using Specman and SystemC with TLM ports. The software is running in the same memory space as the testbench, both of which run through the simulator on the Linux host. The advantage of this approach is fast execution speed of the software and the interoperability of the software with other verification components in the testbench. The software code runs in zero simulation time and the testbench has full control of the software using TLM ports and direct memory access via pointers. In addition, the software code can invoke gdb or any other C debugger to make debugging easier.

Keywords: Specman, System C, TLM, software, hardware, co-verification, ISX, CVL

I. INTRODUCTION

In the past, software usually comes as an afterthought in relation to hardware deliverables and the development of device drivers or firmware happens late in the ASIC product development cycle or even after silicon is available. Imagine delivering silicon to a customer with a register document spanning thousands of pages with untested or absent device drivers or software to abstract those registers. In modern ASIC/SoC designs, software is eclipsing hardware as the main driver of system development cost. Customers now demand that semiconductor companies deliver hardware and software as an integrated and completely functioning system. The quality and the availability of the software becomes a main differentiator in the semiconductor market. Co-verification, that is, testing the hardware and software working together before tape out, is a critical factor contributing to the success of an ASIC/SoC project. [1][2]

Co-verification provides three primary benefits. First, co-verification gives the software engineer early access to the hardware. They can start debugging the software code early in the project without having to wait for the silicon to execute the software on the hardware. By the time the silicon comes back to the lab, there is already working software with most of trivial bugs ironed out. Engineers can focus on testing system integration and use the software to qualify the silicon for production. Thus, co-verification can pull in the project schedule and shorten the time-to-market cycle. Second, co-verification provides additional testing for the hardware design. The verification engineers can use the software executing on the hardware in the simulation environment to generate stimulus that mimics the true operation of the system, which yields better corner cases coverage than using stimulus generated by artificially models from the testbench. Co-verification can identify system integration bugs before tape-out when the bugs are less costly to fix. Third, co-verification provides better visibility in debugging the software and hardware interaction. When the software is running in the silicon, it is a black box system with limited peeks and pokes making it more difficult for verifiers to diagnose bugs in the software and hardware integration. When the software is running in a co-simulation environment, verifiers have white box access to the internal operation of the complete system. Co-verification can increase the productivity of the verifiers in debugging system problems.

In this paper, the authors are searching for a co-verification solution for a new embedded SoC, networking project. Specman and the Cadence simulator are the mainstream tools available to the project team. After evaluated the existing methods available, they decided to develop an alternative co-verification approach that utilize Specman's unique powerful features to overcome the disadvantages of existing methods.

The paper is organized as follows: In Section 2, we introduce a generic SoC testbench architecture and outline the problem statement of co-verification in general. Then we briefly discuss existing methods and highlight their pros and cons. In the subsequent sections, we describe the co-verification approach using Specman and SystemC with TLM ports, with the advantages and challenges of this method.

II. Evaluate Existing Co-verification Methods

A. *Generic Embedded SoC Testbench Architecture*

Figure 1 shows a generic architecture of an embedded SoC and its testbench. The embedded SoC has a CPU, along with network interconnect (NIC) which connects the CPU and various RTL IP blocks via an AXI bus. The RTL IP blocks may consist of data processing units or external interfaces that connect to the pins of the chip. The software executes on the embedded CPU and communicates with the RTL IP blocks via AXI transactions. In the software's point of view, the complex RTL hierarchy of the device is abstracted into a set of accessible register and memory address space.

Our testbench platform is implemented using Specman coupled with UVM (Unified Verification Methodology). The testbench is controlled by a central virtual sequence that co-ordinates the AXI transactions and interaction with the RTL IP blocks. In the absences of the CPU, the AXI UVC (Unified Verification Component) is acting as the BFM (Bus Function Model) of the CPU that drives the AXI transactions sequence from the testbench into the NIC. Coming from a verification background, the most important criteria of co-verification is compatibility with the existing testbench architecture. When executing our testcases, we should able switch between using our own AXI sequences and using the software driver with minimal effort. The software should seamlessly communicate with the other components in the testbench so that we can reuse the scoreboard and checkers in the testbench to avoid re-writing parts of the testbench. Other criteria which are important for a co-verification solution are execution speed and ease of debug, so that we can minimize any productivity hit of testing the software together with the hardware.

Our device uses a commercially available embedded CPU core and RTOS, which minimizes the verification of the low level integration of the software and hardware operation. Our verification focus is on the application layer of the software by testing its functionality and its interaction with the RTL IP blocks. From a hardware verification engineer's perspective, the software is nothing more than a collection of C functions issuing register reads and writes operations into the AXI bus. Thus, the problem statement of co-verification involves answering three simple questions:

1) *How to call a C function from the testbench?*
2) *How can a C function initiate AXI transactions on the AXI bus through the BFM?*
3) *How to check that the AXI transactions and the targeted RTL IP blocks demonstrate the correct behavior?*

In the search for the best technical solution which meets our needs, we investigated three alternatives: acceleration / emulation, ISX/ISS and sockets/CVL. These alternatives are outlined below, before we present the solution which we finally adopted, the Specman to SystemC TLM bridge.

B. *Acceleration/Emulation*

The acceleration/emulation co-verification method uses special equipment to emulate the RTL design. The CPU core is synthesized into the emulation box and the software is running on the CPU cycle by cycle. The Cadence Palladium XP (UXE) [3] acceleration platform supports both In-Circuit Emulation (ICE) and Simulation Acceleration (SA) mode of operation. The advantage of co-verification with the ICE mode is speed of execution, but there is essentially no reuse of the testbench in ICE mode. In SA mode, we can reuse most of the testbench, however the speed drops to that what is imposed by the testbench. In addition, this mode still requires us to integrate the software into the testbench using one of the methods described below. As a result, the software is painfully slow to run even with the 40x simulation speed up in SA mode. Above all, the biggest disadvantage of this method is the price tag, which means the equipment is limited to a very small number of engineers working on the most critical pieces of the system.

C. *ISX/ISS*

This co-verification method uses the Cadence Incisive Software Extension (ISX) technology [4]. ISX provides debug support of the software code running on an RTL model of the CPU core in the simulator or running on a 3rd party Instruction

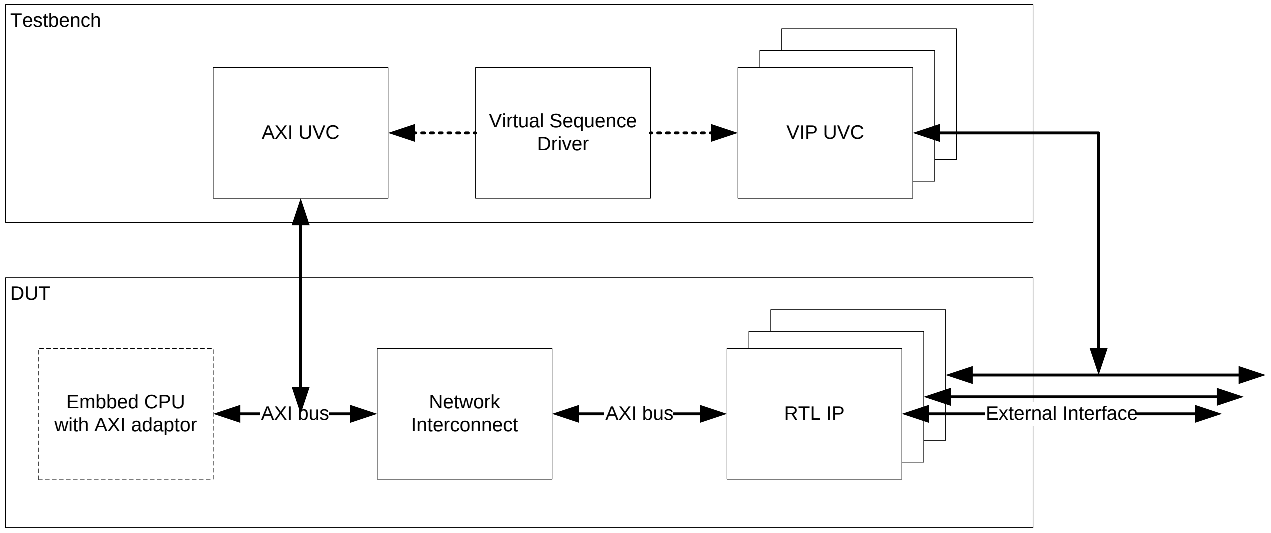

Figure 1. Generic Embedded SoC Testbench Architecture

Set Simulation (ISS) models. ISX integrates with our Specman testbench very well; we can execute the software like it is a native component of the testbench environment. However running the software in SA mode is already painfully slow, running it in pure simulation is a hundred times worse, so it is out of the equation. Using ISX with an ISS model yields good performance by giving up cycle by cycle accuracy. The software code is running in a much faster abstraction model of the CPU and ISX takes care of the AXI bus connections. Although this method is not as expensive as UXE, the use of ISX/ISS still requires purchase of additional licenses. Under some scenarios when we need to test low level integration, ISX/ISS is indeed the best co-verification solution. We still have a hard time justifying the cost of extra licenses when testing high level C code given that the next alternative is free.

D. *Socket/CVL*

This co-verification method uses a UNIX socket communication as an interface mechanism between the hardware and the software [5]. A client server model is implemented to interface the testbench and software via a set of API functions to facility calling C functions from the testbench and calling testbench functions from the software. Specman has built in support of this co-verification method via the Co-Verification Link (CVL) library [6]. The advantage of this method is speed and ease of implementation. The software is compiled as a native executable running on a powerful Linux host, which runs faster than the embedded CPU inside the silicon. Hooking up the testbench to the software only involves setting up the socket connection. The testbench calls external C function like calling native Specman methods. This method almost meets all our evaluation criteria except for two shortcomings. The first problem is the CVL link only support pass by value but not pass by reference. The software cannot access internal data structure of the testbench and vice versa without writing extra CVL API functions. Being able to peek across the language boundary is a very handy feature that allows us to insert customized testbench code into the software to help us debug or carry out white box testing. The second problem is the implementation of Specman CVL requires the testbench relinquish the control to the software. In CVL, the software is a layer above the testbench, so the C code is the master and the testbench is the slave. Although it is relatively easy to write a small piece of code that always passes the control back to the testbench, we believe the software on top architecture is fundamentally wrong conceptually. In a verification environment, the testbench should be the master of everything for efficient testing. It should have absolute control over all the verification components, the RTL code and the software code.

We would have settled for CVL co-verification method if we did not come up with the Specman and SystemC with TLM co-verification method. This method is detailed below.

III. Specman to SystemC TLM Software Bridge

A. *Overview*

Specman has excellent integration support with C code. The Specman C interface is designed for implementing parts of the testbench in C [7]. It provides full access to the Specman memory space in C via an auto-generated C header file which defines all the data structure in the Specman testbench. Specman can call any C function and vice versa through the built-in method interface. However there is a limitation in the Specman-C interface, it does not support TCM (time consuming method) across the language boundary. The C functions can only be executed in zero simulation time. This is a major issue since the software code has to aware of the simulation time because register read and write operations from the CPU to the RTL will cause the simulation time to advance in the simulator.

Consequently, this limitation forced us to investigate SystemC as Specman also has excellent integration support with SystemC code. Specman can interface with SystemC directly using TLM ports via the built in UVM-ML (multi-language) library [8]. A blocking TLM port connects a Specman TCM to a SystemC thread, which understands the notation of simulation time. SystemC is an extension to C++ which is inter-operable with plain C code. In the Cadence simulator, Specman code and SystemC code are executing under the same memory space. By wrapping the Specman-C

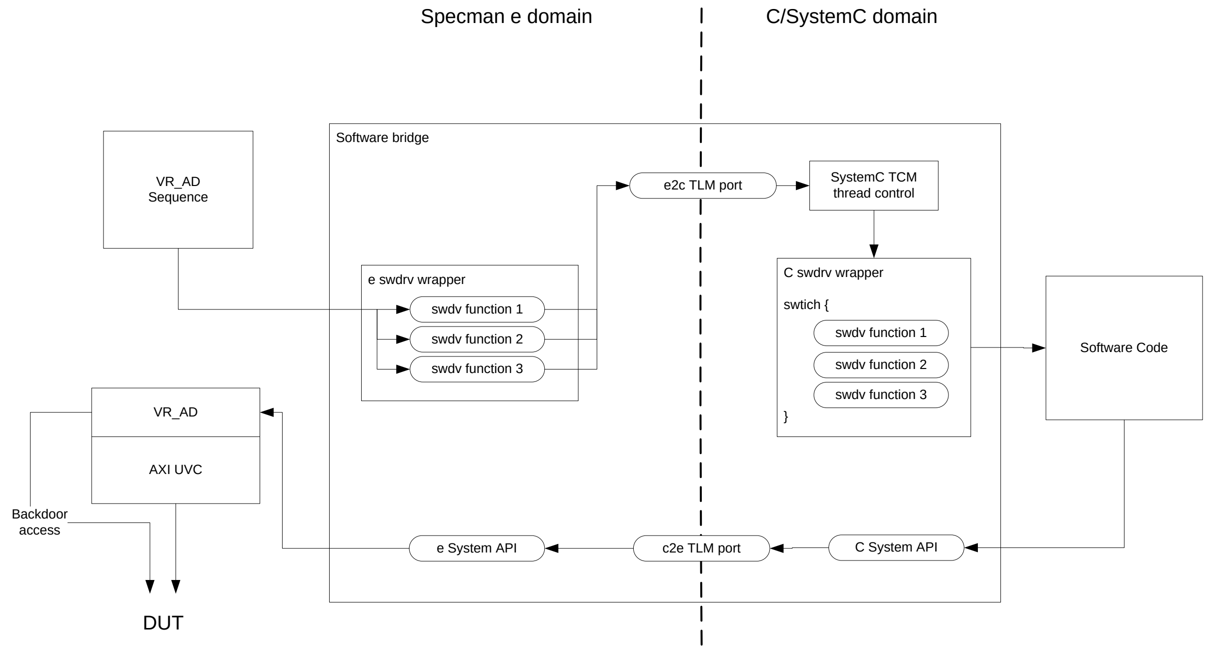

Figure 2. Specman/SystemC TLM Software Bridge

interface inside a SystemC module, we found a way to let C code be aware of the simulation time and invoke Specman TCM methods which initiate the register read/write operations. Thus the Specman – SystemC solution presented the best approach and overcame the limitation inherent with CVL, while not requiring the additional license that the ISX/ISS solution presented.

Given the cross language boundary problem is solved, we developed a Specman/SystemC TLM software bridge shown in Figure 2. The core of the software bridge is pair of blocking put TLM ports; one for the Specman to C calls and other for C to Specman calls. For scalability consideration, we decided to tunnel all function calls into two TLM ports instead of creating separate TLM port for each function. Connecting a TLM port from Specman to SystemC requires a small piece of code (aka, the bridge) in both language domains. It is easier to maintain the bridge by having a fixed permanent connection than keep updating it to support every new function in the software code.

In the following sections, we illustrate the connection inside the software bridge using a simple software function as an example:

```
int foo(int arg1, int arg2, int arg3);
```

B. TLM port and data structure

A TLM port can transport a data structure across the language boundary and we use this data structure to store all the information contained within the function call. The data structure has only two fields, the name of the function and a pointer pointing to the memory address of another Specman data structure which stores all the arguments and the return value for the function call.

```
struct func_call_s {
   func               : func_name;
   arg_ptr            : uint;
}
```

In this example, the function name enum type extension and argument data struct looks like this:

```
extend func_name_t : [foo];
struct foo_arg_s {
   arg1               : int;
   arg2               : int;
   arg3               : int;
   return_value       : int;
}
```

The TLM port will automatically convert the Specman data structure into an equivalent SystemC data structure under the hood. Therefore we have to generate an equivalent data structure of the Specman func_call struct in the SystemC domain. The Cadence UVM-ML library has a handy utility (mltypemap) that auto-generates a SystemC uvm_component class from a Specman struct. With the two matching data structure defined in both language domains, next we have to implement the auto converting function for the TLM port. Since the func_call data structure only has two scalar fields, the converting function is very simple to implement. At last we have to declare the TLM ports in the software bridge:

```
e2c : out interface_port of tlm_blocking_put_if of
(func_call_s) is instance;
c2e : in interface_port of tlm_blocking_put_if of
(func_call_s) is instance;
```

C. Specman to C software functions calls

There are wrapper functions at both sides of the TLM port. On the Specman side, the software bridge has to define a TCM that looks exactly like the software function, in which it fills in the argument data structure and the func_call data structure, and then passes the data structure into the TLM port. Since the software function wrapper appears to be a plain TCM to the rest of the testbench, VR_AD sequence can simply call the software functions like calling any other native Specman TCM without knowing the implementation of the TCM is actually located in the C domain.

```
foo(arg1 : int, arg2 : int, arg3 : int) : int
@sys.any is {
    var foo_arg : foo_arg_s = new with {
        .arg1 = arg1;
        .arg2 = arg2;
        .arg3 = arg3;
   };
   var func_call : func_call_s = new with {
        .func = foo;
        .arg_ptr = foo_arg.get_pointer();
   };
   e2c$.put(func_call);
   return foo_arg.return_value;
};
```

On the C side, the body of the TLM port implementation is a big case-switch statement that unrolls the func_call data structure. First it has to determine which function to call, and then it will resolve the pointer of argument data structure, call the software function with the argument values and store the return value back into the argument data structure. Note that the C domain has full access to the memory address of the argument pointer, so it is possible to support pass by reference arguments in software function call.

```
switch (func_call->func)  {
    case SN_ENUM(func_name_t, foo) :
        SN_TYPE(foo_arg_s) arg = (SN_TYPE(foo_arg_s))
           func_call->arg_ptr;
        arg->return_value = foo(arg->arg1,
            arg->arg2, arg->arg3);
    break;
```

D. C to Specman system methods calls

When the software code needs to issues a read or write operation on the CPU bus, it has to call system wrapper functions. Two of the most commonly use function are system write that write a value to a given address on the CPU bus and system read that fetch the value of a given address from the CPU bus. These two functions will stage the func_call data structure and pass it into the c2e TLM port just like the above examples. Here is an example of the system call wrapper functions:

```
int sys_read(int addr){
    func_call_s func_call;
    SN_TYPE(sys_read_arg_s) sys_read_arg = new;
    sys_read_arg->addr = addr;
    func_call.func = sys_read;
    func_call.arg_ptr = &sys_read_arg;
    c2e->put(func_call);
    return sys_read_arg->return_value;
};
```

In the Specman side, the func_call data structure is unrolled in a similar manner and then calls the corresponding VR_AD operation in the VR_AD sequence driver. The system functions are not limited to system read and write, the testbench can extend the list of the supported system functions depends on the requirement of the software. The following is a list of system functions supported in our software bridge implementation:

1) *sys_write*
2) *sys_read*
3) *sys_read_modify_write*
4) *sys_burst_write*
5) *sys_burst_read*
6) *sys_poll_busy_bit*
7) *sys_wait*

Most of the system functions can be implemented using basic read and write, but sometimes it is more efficient to let the testbench handle some of the repetitive operation to minimize the cross language boundary handshake. The only except is sys_wait, in which the software can indicate it wants to sleep for a period of time, so it pass the control back to the testbench to allow the simulation time to advance.

E. *Software Bridge to Testbench Integration*

The core of the software bridge is implemented in a reusable Specman unit and a SystemC module that contains the permanent binding of the TLM ports that is capable of supporting any software function. The testbench writer has to implement a wrapper function pair in the *e* domain and in the C domain for each of the supported software functions. This is a tedious and repetitive task that is prone to human error. In our current implementation, we automate the generation of the software wrapper. First, we use a document extraction tool, such as Doxygen, to convert function prototypes in the C code to XML. Then we use a Perl script to parse the XML and output the *e* wrapper method and c wrapper function, in which we use the Specman built-in SC2e utilities to auto-generate conversion functions for the data types used in the arguments of the software functions

Most of the software functions are passive functions; they are only executed when they are called by VR_AD sequences in the testcase. However some functions are reactive software functions, such as an interrupt service routine. The testbench has to trigger those software calls upon detecting a change on an interrupt pin in the DUT. The following example illustrates how to hook up the interrupt service route in the software using a Specman event port.

```
interrupt : in event_port is instance;
    keep bind (interrupt,external);
    keep interrupt.edge() == rise;
    keep interrupt.hdl_path() == "dut.int";

event interrupt_triggered is @interrupt$;
```

```
on interrupt_triggered {
    start isr();
};

isr() : int @sys.any is {
    var isr_arg : isr_arg_s = new;
    var func_call : func_call_s = new with {
        .func = isr;
        .arg_ptr = isr_arg.get_pointer();
    };
    e2c$.put(func_call);
};
```

Furthermore, in simulation, when there are lots of register operations in the testcase, although each operation takes a small amount of simulation time, when aggregated they consume the majority of the simulation time. Sometimes it is desirable to speed up the register operation by depositing the value into the register or fetching the value from the registers directly without going through the AXI bus. Since our software bridge implementation connect to the AXI UVC via VR_AD, we can just enable backdoor access in VR_AD to allow all the software write or read transactions to the DUT occurs in zero simulation time to speed up the simulation.

IV. Advantage and Benefits

There are many advantage of the Specman with SystemC over TLM port co-verification method over existing methods. First of all, it is free if the testbench is already implemented in Specman. Our method requires no extra license cost or purchase of expensive hardware equipment. Specman and SystemC are natively supported in the Cadence simulator. The tools used to auto-generate the wrapper functions are either open source (Doxygen) or come with the Specman package (SC2e utilities and UVM library utilities).

Our method has very good execution speed since the software code is compiled and executed on the Linux host alongside with the simulator. Our method is convenient to debug and no extra work is required to hook up a C debugger. The software C code is compiled into SystemC, which allows us to use the built-in SystemC debugger present in the Cadence Simvision tool. We can set break points in the C code, inspect any data structure during function calls and we can even step across the e/C language boundary. Transactions of software function calls can displayed alongside the RTL signals in the waveform viewer using strip chart.

Our method is scalable. Since the software function call is wrapped by a Specman TCM method, we can easily swap in a VR_AD sequence to replace the software function call using a WHEN subtype of the wrapper function. It allows the verifiers to bypass the software code temporary if the software becomes a road block to verifying the DUT. We can also swap in ISX for cycle accurate simulation using the golden RTL model of the CPU in order to verify the low level system integration. The wrapper function TCM can have another WHEN subtype that passes the function call into ISX's GSA (generic software adapter) interface.

Our method can also provide coverage on software function. Since all of the software function arguments are encapsulated in a Specman struct and all the function names are listed in an enum type, we can easily generate coverage groups on the software function arguments within the automated wrapper generation script. The software bridge will emit a coverage event every time a software function is called. We can also use the VR_AD built-in coverage to collect coverage of the register and memory space addressed by the software functions.

V. Challenges and Future Development

The biggest challenge of this co-verification is debugging errors in the C code that are related to pointer handling, which has the effect of crashing the simulator. Since the simulator and the software are running in the same memory space, segmentation fault in the software can bring down the simulator and bad pointer assignments can corrupt memory in the simulator kernel or Specman engine which often results in a core dump. Pointer problems are the most common and most nasty source of errors in C programming and it is both the power and the weakness of the language. There is nothing much the verifiers can do since the root cause of the problem is bad C code. In our co-verification guidelines, we require that the software team thoroughly tests their C code before integrating with the hardware in co-verification. The use of memory debugging tools, such as Valgrind, is advised to make sure there is no memory problem in their code before releasing their code over to the verification team.

The second challenge of this co-verification method is debug turnaround time of the software C code. Since the C code is compiled into SystemC, which is compiled and statically linked to the RTL simulation snapshot during the elaboration phase, we cannot fix the bugs in the C code in the middle of simulation without going through the whole RTL elaboration process. In the future, we plan to use dynamically loaded shared library to resolve this problem. Instead of statically linking the software C code into the SystemC software bridge, we can compile the software C code into a shared object. Then the SystemC software bridge can use the Linux built-in dynamic linking loader library to dynamically load the software shared object and obtain the address of the symbols of the software functions at the beginning of the simulation. If a bug is identifier in the C code during simulation, the verifier can manually drop the loaded software image, fix the C code and recompile the shared object, then reload the software image without quitting the simulator.

In addition, the current implementation of the software bridge for this co-verification method only supports Specman based testbenches. We have plans to enhance the software bridge to support this co-verification method for System Verilog testbenches. However there are several technical challenges we have to overcome to make the System Verilog implementation a reality. Both Specman and System Verilog support TLM port connection to SystemC under UVM, but native TLM port binding can only support pass by value. One alternative for the System Verilog implementation is giving up the support of passing by reference and only supports the use of pass by value in the function arguments. Another alternative is using VPI (Verilog Procedural Interface) to manipulate pointers of System Verilog objects directly in the software bridge. Given that the problem of passing data type between the System Verilog and C language boundary can be resolved, we still have to implement the equivalent of the Specman utilities library for automated type conversion in System Verilog for the software function wrapper generation script. In theory it is feasible to port this co-verification method to System Verilog, however it requires more research to prove its practicality.

VI. Conclusion

In this paper, the authors developed an alternative methodology for hardware software co-verification using standard verification language and industry interface standards. There are many advantages of this co-verification method including the low cost, the fast execution speed, the transparent visibility in debug and the ease of use when setting up the environment. This allows our existing Specman testbench evolve into a hardware software co-verification platform. The software team gain early access to the hardware design in order to test the software and hardware integration issues and the verification team has more accurate stimulus generated from software operation for testing the RTL code.

Acknowledgment

Thanks to Sebastien Regimbal, Grant Brydon and Gordon Bastien for their contributions in the development of the software bridge.

References

[1] A. Jason, “HW/SW co-verification basics,” EE Times, 2011.

[2] D. Rittman, “Hardware/Software Co-verificatoin & Co-Simulation,” 2004.

[3] Cadence, “Palladium XP (UXE) User Guide.”, 2010

[4] Cadence, “Incisive Software Extension (ISX) User Guide.”, 2011

[5] A. Freitas, “Hardware/Software Co-Verification Using the System Verilog DPI,” DASS, 2007.

[6] V.G. Kumar, “A New Methodology for Hardware Software Co-verification,” IPSOC, 2006.

[7] E. Zwingenberger, “A Simple New Approach to Hardware Software Co-Verification,” EE Times, 2007.

[8] Cadence, “Universal Verification Methodology (UVM) Multi-Language Methodology”, 2010

[9] Cadence, “Specman Usage adn Concepts Guide for e Testbenches.”, 2010

[10] Cadence, “Specman Integrators Guide.”, 2010