The Integration of SystemC and Hardware-assisted Verification

Ramaswamy Ramaswamy and Russell Tessier

Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003
{rramaswa,tessier}@ecs.umass.edu

Abstract. In this research a refined interface between high-level design languages and hardware verification platforms is developed. Our interface methodology is demonstrated through the integration of a communication system design, written in C and SystemC, with a multi-FPGA logic emulator from Ikos Systems. We show that as designs are refined from a high-level to a gate-level representation, our methodology improves verification performance while maintaining verification fidelity across a range of abstraction levels.

1 Introduction

C-based system level design environments, such as SystemC [6], have recently been introduced to allow for modeling of entire systems in high-level language. C-based hardware modeling, enabled with the use of C++ based class libraries, allows for concurrent verification of both system hardware and software and the definition of interfaces between them. Although software can efficiently be tested using processor-based tools, the latter stages of hardware development frequently involve the use of verification hardware, such as parallel simulators or logic emulators. Traditionally, the interface between software test environments and hardware verification has, at best, been inefficient. Although it is currently possible to integrate software written in high-level languages, such as C/C++, with hardware descriptions written in HDLs, such as Verilog and VHDL, the overhead of passing data between the two verification domains can be a bottleneck, limiting verification performance [5]. The need to integrate system-level software (SystemC) with verification hardware motivates a new, modular integration approach.

In this paper, a design methodology is described which allows for the integration of parallel logic verification equipment with C/C++ system design languages. Our approach [4] *isolates* individual SoC component models from standard SystemC inter-component interfaces. Both component logic and interfaces are structured to allow for straightforward update as individual portions of the design are refined. This methodology allows for the optimization of the verification hardware interface to SystemC and similar system-design languages. Specifically, hardware verification approaches involving transaction-based processing [3] and data buffering can be accommodated to provide a transition from software to hardware domains. Communication support between components is provided at the functional, bus-cycle accurate, and cycle-accurate levels.

Our approach is demonstrated by the integration of SystemC with a VirtuaLogic emulation system [2] from Ikos Systems. To illustrate the system's capabilities for SoC designs, a modular communication system design is verified using SystemC with the FPGA-based emulator. Initially, the entire design is modeled in software using SystemC. After logic component and on-chip communication refinement, a portion of the design is migrated to the emulator. We show that it is possible to achieve increased verification accuracy with the use of our integration methodology over a range of modeling abstraction levels (from behavioral to gate-level). Additionally, we show that the interface approach is superior to previous process-based tool interfaces, such as PLI, used by logic simulators.

2 Background

Traditionally, it has been difficult to verify entire systems at a cycle-accurate level using a high-level language. Often, a complete design re-write has been necessary to translate behavioral portions of a hardware design into a format that can be synthesized to hardware. *SystemC* [6] provides high-level support for cycle-accurate hardware through the use of a set of C++ class libraries and a simulation kernel that supports clock-based hardware modeling. The result of this specification is the standardization of all design information, the capability to quickly re-specify and evaluate design changes, and the ability to increase overall verification speed compared to coupled high-level language/HDL simulator approaches. The SystemC class library provides special support for process concurrency and clocked hardware evaluation [6].

Current approaches for integrating logic simulators with C-based designs are too slow and inefficient for parallel verification. For simulators, data is passed between language domains by means of remote procedure calls or inter-process communication approaches such as sockets [5]. Emulation generally requires special operations regarding support for multi-cycle data transactions [3] and data buffering to support fast data rates. Further complicating the integration of SystemC and verification hardware is the design evolution of most SoC components. Most designs require the development of both logic functionality (IP cores) *and* inter-core interfaces. Not only does this require validation and refinement of the logic at behavioral, RTL, and logic levels, but it also requires the gradual refinement of the protocols necessary to connect these modules. The capability to verify communication using abstract ports (functional), untimed bus protocols (bus-cycle accurate), and clock-based protocols (cycle-accurate) is an important aspect of SystemC [1] [6]. The component isolation offered by these representations allows SystemC to model hardware accurately at various stages of the design cycle. Additionally, component isolation allows for a framework in software for emulator interfacing.

SC_MODULE - Basic object definer for SystemC objects.
SC_CTOR - Constructor used for initializing signals and declaring process types.
SC_METHOD - Instantiates a function which executes in zero functional time.
SC_SLAVE - Indicates slave process will start when value received on input port.
sensitive() - When the value of the enclosed signal changes, the process executes.
wait() - Suspends execution of a process until sensitivity signal changes.
sc_in, sc_out - Specifies input and output ports.
sc_outmaster - Output port of a master process.
sc_inslave - Input port of a slave process.

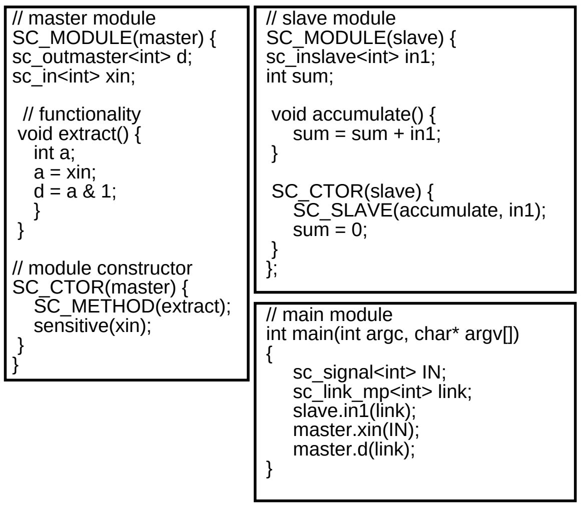
Fig. 1. SystemC Terminology

3 Integration Approach

Our design methodology can be demonstrated through a series of code examples. System design languages, such as SystemC, contain a range of constructs, shown in Figure 1, which can be used to define functionality and component interfacing. In Figure 2, a set of *untimed functional* (UTF) modules are shown. The UTF level of functional abstraction provides for the highest-level specification of a system. The example system consists of modules **master**, **slave** and **main** that communicate over *functional* channels in a sequential fashion. Data operation and execution order are modeled accurately, but time is not. All processes execute in zero time. In SystemC, the internal, *untimed* functional model of each module can be refined to either a *timed* functional or *cycle-accurate* model without modifying the module port structure or surrounding modules which interface with the module.

Communication between the two child modules in Figure 2, **master** and **slave**, takes place via the SystemC master-slave library [6]. Systems that contain DSPs, custom ASIC cores and processor cores communicating over a set of buses can be modeled with library structures as an interconnection of sequentially communicating blocks. When **master** writes a value to its output port through the *extract()* process, the *accumulate()* process in **slave** is invoked. Master output port (*sc_outmaster*) and slave input port (*sc_inslave*) are linked by an abstract channel, defined by the SystemC keyword *sc_link_mp*. A write to an output port starts the second slave function through *sc_link_mp*. The slave process executes inline with the master process and returns control to the master process after execution. This approach is amenable to communication synthesis. The top-level module (**main** in Figure 2) indicates connectivity of communication.

SystemC provides two levels of inter-component communication, *functional* and *bus-cycle accurate* (BCA). Migration from functional to BCA communication takes place in conjunction with migration of logic functionality from untimed functional to gate-level representation. The bus-cycle accurate definition specifies inter-component interaction at the cycle-accurate level. Communication



```
// master module
SC_MODULE(master) {
sc_outmaster<int> d;
sc_in<int> xin;

  // functionality
 void extract() {
    int a;
    a = xin;
    d = a & 1;
    }
 }

// module constructor
SC_CTOR(master) {
    SC_METHOD(extract);
    sensitive(xin);
 }
}
```

```
// slave module
SC_MODULE(slave) {
sc_inslave<int> in1;
int sum;

 void accumulate() {
     sum = sum + in1;
 }

 SC_CTOR(slave) {
     SC_SLAVE(accumulate, in1);
     sum = 0;
 }
};
```

```
// main module
int main(int argc, char* argv[])
{
     sc_signal<int> IN;
     sc_link_mp<int> link;
     slave.in1(link);
     master.xin(IN);
     master.d(link);
}
```

Fig. 2. UTF Master-Slave Module

between SoC components is made cycle-accurate with respect to bus handshake protocols while component functionality is unchanged. Abstract ports such as *sc_outmaster* and *sc_inslave* are adapted to form bus ports; hierarchical entities that group together specific terminals for data, address and control signaling. Three bus protocols are supported: no-handshake, enable-handshake and full-handshake. User-defined protocols can also be established.

Figure 3 shows how functional communication through abstract channels at the UTF level can be refined to a full-handshake bus protocol. Refinement is accomplished through the use of protocol conversion modules, *abs2full* and *full2abs*. In the full-handshake protocol, the ports have three terminals - *data*, *req*, and *ack*. Each data transfer cycle proceeds as a sequence. The *abs2full* module at the data sender asserts *req* and places the data to be transfered on *data*. When the *full2abs* module receives the data, it asserts *ack*. When *abs2full* receives the *ack* signal, the next data item can be transferred. The use of such protocol conversion modules separates component functionality from inter-component communication. This provides a pluggable environment where different modules implementing functionality and communication can be swapped easily.

The Transaction Interface Portal (TIP) [2] from Ikos Systems is a verification environment that enables a C model running on a host workstation to communicate with an RTL model implemented on the emulator. This capability provides a verification methodology called *co-modeling*. A common application of co-modeling is verification of a design under test (DUT) which is implemented on the emulator. A testbench or supporting system model is implemented as a C application running on the host workstation. Driver software coordinates data transfer with the DUT running on the emulator. The Transaction Application Programming Interface (TAPI), a series of C drivers, is used to control

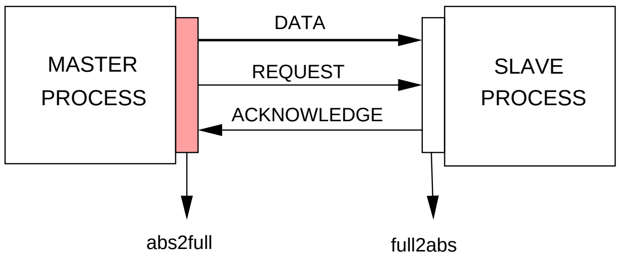


Fig. 3. Protocol conversion modules integrated with master-slave modules

```
// master module
SC_MODULE(master) {
sc_outmaster<int> d;
sc_in<int> xin;

  // functionality
 void extract() {
    int a;
    a = xin;
    d = temp & 1;
    }
 }

// module constructor
SC_CTOR(master) {
    SC_METHOD(extract);
    sensitive(xin);
 }
}
```

```
// slave module
SC_MODULE(slave) {
sc_inslave<int> in1;
int sum;

  void accumulate() {
     tapi_enable();  // open emulator
     tapi_wr_construct; // build write object
     tapi_write(in1);
     tapi_rd_construct; // build read object
     tapi_read(sum); // get result
 }

 SC_CTOR(slave) {
     SC_SLAVE(accumulate, in1);
 }
};
```

Fig. 4. Modified SystemC Slave

the operation of the emulator. A workstation-based PCI card provides physical communication between the workstation and the emulator.

The TIP architecture can be used in one of two ways - *data streaming* or *reactive co-modeling*. In data streaming, data transfers are independent of each other and allow for constant interaction between the user application and DUT. In reactive co-modeling, the data transfer sent by the user application depends on the previous transfer. The user application waits for the DUT to process the current transfer before a new one is sent, potentially leading to an application idle period.

SystemC modules can be modified to allow for emulator calls. In the example shown in Figure 4, a series of TAPI driver calls allow for a software-hardware interface for the UTF **slave** function shown in Figure 2. The inter-component communication infrastructure remains the same as for software-only verification. The isolation of inter-component communication supported by SystemC provides an ideal interface for parallel verification hardware. By taking advantage of this isolation, a number of optimizations for emulation can be supported.

Logic emulator interfaces often require special synchronization techniques to allow for efficient data transfer. Event-based and cycle-based synchronization are examples of fine grained synchronization in which the verification platforms synchronize at every event and clock cycle, respectively. Due to this tight coupling, the entire system proceeds at the rate of the slowest domain, limiting performance. An alternative approach is to synchronize data transactions only when necessary via *transactions* [3]. A transaction is a multi-cycle communication sequence between two verification domains. Transactions contain both data and synchronization information. A single transaction results in multiple verification cycles of work being performed by a verification platform (logic emulator). The transaction can be as simple as a memory read or as complex as the transfer of an entire structured packet through a channel.

4 Experimental Methodology

To evaluate our approach of integrating SystemC and hardware-assisted verification, the functionality of two testbench designs were verified using combinations of SystemC, logic simulation, and a logic emulator. All software tests were performed on an unloaded 360 MHz Sun Ultra 60 workstation with 512 MB of RAM. The workstation interfaces to an Ikos VirtuaLogic VLE-2M logic emulator containing 128 Xilinx 4036EX FPGAs via an SPCI card. The emulator clock speed was set to 30MHz for all designs. Data transfer between the workstation and the emulator was performed through the use of data streams and data transactions.

Two cores, a Reed Solomon encoder/decoder core (51,825 gates and 1,233,397 vectors) and a palindrome detector circuit (13,577 gates and 200,000 vectors) were used to validate the functionality of the SystemC-emulator interface.

Each of these cores was verified in three formats:

1. The cores were first modeled in functional SystemC code and compiled using **gcc** and SystemC library version 1.2 [6].
2. A gate-level model of each core, created from synthesized RTL models of the cores, was simulated using the Cadence Affirma NC-VHDL tool set.
3. A gate-level model of each core was mapped to the emulator using the Ikos VirtuaLogic compiler [2].

To fully test the interaction of the cores in a heterogeneous verification environment, two test scenarios were created. For the commercial *Reed Solomon* coder, a testbench written in SystemC code was created from the vectors obtained from the commercial core vendor. The testbench code was modeled separately at untimed, timed, and cycle-accurate levels, as described in Section 3. The three versions of the testbench were interfaced with three different implementation versions of the cores (SystemC, simulated gate-level, emulated gate-level) via functional and bus-cycle accurate methods of communication. This provided the capability to measure transfer rates between software and hardware verification tools under differing accuracy levels. The vectors used with the Reed Solomon coder are representative of the effort needed to decode an eight bit 200 x 200 Portable Greymap (PGM) image. To further test the Reed Solomon coder in an *integrated* system environment, an entire communication system, including the Reed Solomon core as a critical component, was modeled in software. This

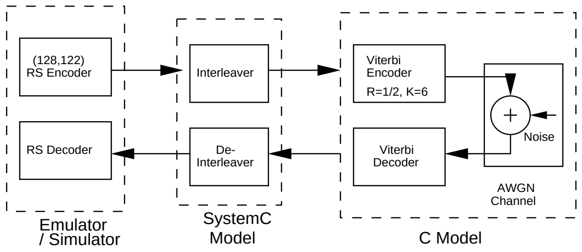


Fig. 5. Modeled communication system

system, shown in Figure 5, consists of the Reed Solomon coder, interleaver/de-interleaver functions [7] and a Viterbi coder.

The variety of components associated with the system makes it ideal for testing SystemC integration. Interleaving and de-interleaving was performed using SystemC models. The Viterbi portion of the system was implemented as a C model. The interleaver, de-interleaver and Viterbi portions of the system were run on the host workstation as thread processes which communicate with each other via abstract master-slave ports. In separate experiments, the Reed Solomon encoder/decoder was modeled using SystemC at the functional level and on the Affirma simulator and VirtuaLogic emulator at the gate level. Communication with the Reed Solomon coder is through PLI based socket calls for simulation and via transaction-based processing for emulation. A similar testbench-based environment was also created for the *palindrome* circuit. Several implementations of the testbench in varying accuracy levels were created and interfaced to the modeled core using functional and bus-cycle timing.

5 Experimental Results

To validate our approach of isolating emulation resources with modular communication constructs, three sets of experiments were performed with the experimental methodology described in Section 4. In the first experiment, a direct comparison of execution time of verification environments which include only SystemC with those that include simulation and emulation is provided. Table 1 shows the times taken to verify the three test configurations. For *Reed-Solomon* and *palindrome*, the test cores were interfaced to testbenches written in untimed functional SystemC, as described in Section 3. For *RS System*, the Reed Solomon core was interfaced to a software version of the communication system described in Section 4. Results in the Table 1 include:

– In the *SystemC* configuration (row 2), the entire design along with the testbench is implemented in SystemC as untimed models.
– In the *SystemC+Sim.* configuration (row 3), the untimed testbench used in the previous configuration is coupled with an RTL description of the DUT running on a simulator. Both testbench and simulator run on the workstation

	Reed Solomon	Palin-drome	RS System
SystemC	0.09s	0.4s	0.34s
SystemC+Sim.	2190s	312s	93s
SystemC+Emul.	175s	16s	43s

Table 1. Verification times

Palindrome Detector				
Data Transfer	Time (sec)		Transfer Rate (kbps)	
Abstraction Level	Simulation	Emulation	Simulation	Emulation
Untimed Functional	285	16	44.91	800.00
Timed Functional	291	19	43.98	673.68
Bus Cycle Accurate	301	24	42.52	533.33
Cycle Accurate	328	29	39.02	441.38

Reed Solomon Coder				
Data Transfer	Time (sec)		Transfer Rate (kbps)	
Abstraction Level	Simulation	Emulation	Simulation	Emulation
Untimed Functional	2260	175	52.39	676.61
Timed Functional	2447	204	48.38	580.42
Bus Cycle Accurate	2524	266	46.91	445.14
Cycle Accurate	2649	290	44.69	408.30

Table 2. Verification times with data transfer modeled at various levels of abstraction

as distinct processes. Communication between the processes is done via PLI-based socket interfaces.

– In the *SystemC+Emul.* configuration (row 4), the untimed testbench, running on the workstation, is coupled with the benchmark core implemented on the emulator.

It can be seen that the same modeling fidelity can be preserved by transitioning from a SystemC model to an implementation on the emulator. Although gate-level emulation takes longer compared to behavioral SystemC verification, accuracy for the cores is enhanced.

In a second experiment, verification times for SystemC testbenches interfaced to cores at various data transfer abstraction levels were determined. In the experiment, testbenches were interfaced to gate-level cores modeled on the emulator and simulated with the NC-VHDL simulator. For each configuration, master-slave interfaces were described in SystemC at various levels of abstraction. Overall run time for the *palindrome* and *Reed Solomon* benchmarks are shown in Table 2 and were measured using *gprof* and the profiling option in the Affirma simulator. Simulation-based verification is significantly slower than emulation-based verification due to the overhead of PLI calls and the sequential nature of execution. Transfer rates indicate achieved data rates between the

Test Bench	Number of vectors	Workstation (sec)		Emul. (usec)
		Verify	Compare	
T1	61714	0.499	1.4	795
T2	68066	0.538	1.6	612
T3	128270	1.022	2.65	621
T4	170594	1.34	3.45	706
T5	179804	1.41	3.65	791
T6	275262	2.15	5.55	-

Table 3. Times taken for emulation with the testbench on the host workstation and the emulator

SystemC testbench and NC-VHDL (simulation) and the emulator (emulation). Columns 4 and 5 show the transfer rate across the interfaces for different levels of abstraction. The transfer rates become slower at lower levels of inter-component communication abstraction. Moving from untimed functional to cycle-accurate modeling offers increased modeling accuracy at the cost of longer verification time. The variation in transfer rate is more noticeable for emulation.

A significant portion of verification time is spent in transferring fixed test vectors between software and hardware verification domains. In a final experiment, testbench vectors, which were previously implemented in SystemC on the host workstation, were migrated to memory resources located on the logic emulator. The testbench located on the emulator was partitioned into two separate memories. One portion contained the input test vectors and the other portion contained the expected output vectors. The test commences when the workstation sends a signal to emulator indicating that vector sequencing should start. Subsequently, individual test vectors are applied sequentually to the emulated design and results are collected and compared to expected vector outputs. After the final vector, a pass/fail result is returned to the workstation. A pass result is sent if all output vectors match expected results.

The above method was implemented with the Reed Solomon coder for varying testbench sizes. Table 3 compares results of storing test vectors on the workstation versus migrating the vectors to the emulator. The numbers in the third column represent the time taken to send and receive the entire set of test vectors to the emulator by a testbench on the host workstation. The fourth column represents the time taken to compare output result vectors with expected output vectors. This comparison was performed by a C program on the host workstation. The last column represents the verification time when the testbench is entirely implemented on the emulator. This also includes the time taken to compare output result vectors with expected output vectors. It can be seen that the verification performance when the testbench is migrated onto the emulator is 5000 times faster on average, than when the testbench is located on the host workstation and transmitted to the emulator. All run times were measured with *time()* function calls in the user application.

The number of test vectors that can be stored on the emulator depends on the amount of free memory available on the emulator. The largest testbench, T6, did not fit on the emulator due to a lack of memory in the system.

6 Conclusions

In this paper we have outlined a new design methodology for integrating system-design languages, such as SystemC, with parallel verification hardware. By isolating the interface to a specific module, optimizations such as data buffering, testbench migration, and transaction-based data transfer can be supported for logic emulation. To overcome data transfer bottlenecks, it was possible to seamlessly migrate benchmark data across the workstation/emulator interface to the emulator. These approaches led to an improvement in verification time while maintaining support of existing inter-component interfaces in software and associated benchmarks

7 Acknowledgments

This work was supported by Ikos Systems, Texas Instruments, and National Science Foundation Grant CCR-0081405.

References

1. K. Bartleson. *A New Stardard for System-Level Design*. Synopsys, Inc., 2000. http://www.systemc.org/.
2. Ikos Systems, Inc. *VirtuaLogic VLE-5 Emulation System Manual*, Jan. 2001. http://www.ikos.com/products/vsli/index.html.
3. M. Kudlugi, S. Hassoun, C. Selvidge, and D. Pryor. A Transaction-Based Unified Simulation/Emulation Architecture for Functional Verification. In *ACM/IEEE Design Automation Conference (DAC)*, June 2001.
4. R. Ramaswamy. The Integration of SystemC with a VirtuaLogic Emulation System. Master's thesis, University of Massachusetts, Department of Electrical and Compter Systems Engineering, September 2001. http://www.ecs.umass.edu/ece/tessier/systemc-thesis.pdf.
5. L. Semeria and A. Ghosh. Methodology for Hardware/Software Co-verification in C/C++. In *Asia and South Pacific Design Automation Conference*, Jan. 2000.
6. SystemC. *SystemC 1.2Beta User Guide*, 2000. http://www.systemc.org.
7. S. Wicker. *Error Control Systems for Digital Communication and Storage*. Prentice Hall, Edgewood Cliffs, N.J., 1994.