Methodology for Hardware/Software Co-verification in C/C++

Luc Séméria
lucs@azur.stanford.edu
Computer System Lab.
Stanford University

Abhijit Ghosh
ghosh@synopsys.com
Synopsys Inc.
Mountain View, CA

ABSTRACT

In this paper we present our C/C++-based design environment for hardware/software co-verification. Our approach is to use C/C++ to describe both hardware and software throughout the design flow. Our methodology supports the efficient mapping of C/C++ functional descriptions directly into hardware and software. The use of C/C++ to model all parts of the system provides great flexibility and enables faster simulation compared to existing methodologies. We show how co-simulation is done efficiently and effectively at the various levels of abstraction and give an example of implementation for the 8051 architecture.

1. INTRODUCTION

With shrinking device sizes, microprocessors, digital signal processors, memory and custom logic are being integrated into a single chip to form systems-on-chip. Verification of such systems poses unique challenges because unlike systems-on-board, in-circuit emulators cannot be used and internal wires are not accessible.

In a traditional design methodology, hardware and software design takes place in isolation with the hardware being integrated with the software after the hardware is fabricated. Bugs that cannot be fixed in software lead to costly re-fabrication and can adversely affect time-to-market.

To avoid costly silicon re-spins and improve time-to-market, the design methodology has to change such that hardware and software are integrated earlier in the design-cycle. Hardware-software co-verification technology is the enabler for this new design methodology.

Hardware-software co-verification involves the simulation of a processor model with a simulation of the custom hardware usually described using Hardware Description Languages (HDL). There is an overhead in passing data back and forth between the largely HDL-based hardware world and the largely C/C++-based software world during co-simulation. This overhead can be reduced [4], but cannot be eliminated altogether.

Recently, several C/C++-based hardware design tools and methodologies have been presented [3]. A C/C++-based design methodology has several advantages over an HDL-based methodology. In a C/C++-based methodology, designer productivity can be improved significantly because one can eliminate translation from a C/C++ system specification to an HDL specification for implementation, by synthesizing directly from the C/C++ specification. This not only reduces translation time but eliminates bugs introduced during translation, which can take significant time to track down. In addition, one can also ease the verification bottleneck by reusing the testbenches that were developed during system validation. In addition, a C/C++-based methodology enables hardware-software co-design and gives designers the ability to perform hardware-software co-verification and performance estimation at very early stages of design.

In this paper we show how hardware-software co-verification is performed in a C/C++-based flow. Our approach is to use C/C++ to describe both hardware and software. In particular, we will use the SYSTEMC (formerly known as SCENIC) environment [8,14] throughout the design flow. The use of C/C++ to model all parts of the system provides great flexibility and enables faster simulation compared to existing methodologies.

The contributions of this paper are to show how co-verification can be done efficiently and effectively in a C/C++-based design environment, how various levels of abstraction are supported, the advantages of a C/C++-based flow from the verification point of view, and how co-verification can be used to drive co-design through performance estimation.

The rest of this paper is organized as follows. In Section 2, we present previous work in this area. In Section 3, we give an overview the SYSTEMC class library used to model hardware in C/C++. In Section 4, we present our design flow. In Section 5, we present different CPU models and techniques to speed-up simulations. Finally in Section 6, an example of implementation for the dw8051 architecture is given.

2. PREVIOUS WORK

Different approaches have been presented for hardware-software co-simulation. Most previous approaches emphasize co-simulation between an HDL and a C/C++ world. POSEIDON [6] performs concurrent event-driven simulation of multiple software or hardware modules. For heterogeneous environment such as the co-simulation of C and Verilog HDL or VHDL, the communication between software and hardware can be done using remote procedure calls [1,15] or some form of interprocess communication (sockets). These techniques have been used in commercial tools such as Synopsys EAGLE [13] and Mentor Graphics SEAMLESS [9].

In [11], a review of the different level of abstraction used to validate a system during the various phases of the design process is presented. Processor models come in various flavors. Bus-functional models (BFM) implement only the bus functionality of the processor and are independent of the representation of the CPU. Instruction-accurate models can be used to simulate accurately instruction fetch, decode, and execute units in a processor. Cycle-accurate processor models can be used to accurately simulate processor behavior at the cycle-accurate level. Instruction-accurate level and cycle-accurate models usually contain a bus-functional model so that they can be used for co-verification.

In [4,9], several techniques are used to speed up simulation by optimizing instruction fetch and memory transaction and suppress-

ing clock on specific modules. These techniques are fairly universal and can also be exploited in our environment.

There are few purely C/C++-based approaches to co-simulation. In [10], the PTOLEMY framework is used to simulate a system. CoWare N2C [2] supports hardware/software co-simulation at different steps of the design. Our work can be distinguished from the PTOLEMY and the CoWare systems by offering a very flexible environment which enables the development of optimized models for both synthesis and simulation at the different stages of the design process directly in C/C++. In particular, our methodology supports the efficient mapping of C/C++ functional descriptions directly into hardware and software.

3. THE SYSTEMC ENVIRONMENT

C/C++ are software programming languages and have little support for describing hardware efficiently. To model hardware in C/C++, we need the following language features that are not present in C/C++.

Concurrency: hardware is inherently parallel, while C/C++ programs are inherently sequential. The SYSTEMC environment introduces the notion of processes, which encapsulates programs that execute concurrently. A system is described as a network of processes and the semantics of process execution is akin to that of communicating sequential processes. Instead of extending the language with new syntactic constructs, concurrency is encapsulated in an object or class. One can build hardware processes by using the subtyping and virtual function facilities of C++.

Signals: hardware processes need to use signals (akin to wires or buffered channels) to communicate with one another. In the SYSTEMC environment, signals are modeled using template classes.

Reactivity: hardware systems are in continuous interaction with their environment, i.e. they are reactive. The notion of reactivity is essential to describing hardware systems at all levels of abstraction. Reactivity is implemented in the SYSTEMC environment through a hybrid of event-driven and process-driven approaches. More details can be found in [8].

Data abstraction: C/C++ supports data abstractions that are useful for software programming. However, for hardware, one needs arbitrary precision signed and unsigned integers, bit vectors and fixed point types. In the SYSTEMC environment, these types are provided as classes.

The SYSTEMC environment can be used to describe an entire system efficiently (even analog parts). A complete description of this environment is beyond the scope of this paper and can be found in [14]. An example of synthesizable hardware modeled in C++ can be found in [5].

4. DESIGN FLOW

The design process starts with the designer creating a functional specification of the system. The aim is to validate the algorithms and system functionality. The functional specification is a network of processes communicating through signals. The processes represent functionality and may need to be mapped to different architectural blocks to be implemented in software or hardware. Therefore, once the system functionality is validated by simulating the specification (in this case, compiling and executing the C++ program), the functional specification is mapped into an architectural specification.

In the architectural specification, processes represent actual hardware blocks, like a processor, memory and ASIC. Communication between processes is performed using signals that represent actual communication resources available in the architecture. It is in the mapping step that the functional specification is partitioned into hardware and software components. Using one language for the description of both hardware and software makes mapping easier, allows the designer to move functionality from hardware to software and vice versa easily, thereby allowing the exploration of different architectures and partitions between hardware and software.

The architectural specification may include one or more processor blocks and other hardware blocks (like DMA, memory, etc.) After the architectural specification is created, the specification needs to be simulated to validate the architecture and determine its performance. This is where hardware-software co-simulation first comes into the methodology. Since various architectures may need to be explored through simulation, co-simulation at this level needs to be extremely fast. Therefore, the models used at this level tend to be more abstract. For processors, only a bus-functional model (BFM) is used. For the other hardware blocks, abstract models with proper interface behavior are used.

Once an architecture is decided, the individual hardware and software blocks are refined by adding the necessary implementation details and constraints for synthesis. Since the hardware and software teams work separately and in parallel, co-simulation has to be used constantly to ensure that the system still works. Since the hardware interfaces have been refined and the software has been targeted to a processor, models used are more detailed and therefore simulation is slower. For processors, a BFM has to be used in conjunction with an instruction set simulator (ISS). For other hardware blocks, register-transfer level (RTL) or behavioral implementable models have to be used.

After this point, hardware can be implemented using synthesis tools and compilers can be used for software. Since this part of the design flow is well established, we will not describe it any further. We would just like to point out that after synthesis, i.e. at the gate-level, whatever co-simulation techniques are in use today can be employed.

5. PROCESSOR MODELS

5.1 Bus Functional Model

A bus functional model of a processor encapsulates the bus functionality of a processor (see [7] for a definition). Such a model can only execute bus transactions on the processor bus (with cycle accuracy), but cannot execute any instructions. A BFM is therefore an abstract processor model that can be used to verify how a processor interacts with its peripherals.

A BFM is a key component in any co-verification solution. In our design methodology a BFM is used throughout the design process. In the early stages of the design process, only the BFM is used for co-simulation, as shown in Figure 1. The BFM provides a pro-

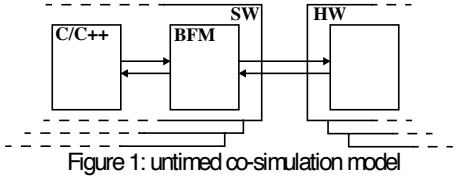

Figure 1: untimed co-simulation model

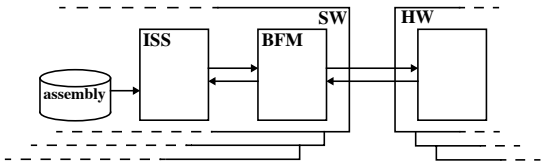

Figure 2: cycle-accurate co-simulation model

gramming interface that can be used by the software directly. Since the software runs on the host processor (on which development is done), this model is untimed because the software execution time is not accurate.

In the later stages of the design process, an ISS needs to be used in conjunction with the BFM in order to execute the instructions for the target processor (Figure 2). The ISS makes use of the same programming interface to talk to the BFM. Since this ISS can be cycle-accurate, one can perform cycle accurate simulation at this stage.

In this section we will present how the SYSTEMC environment is used to develop BFMs and verify hardware-software communication in C/C++.

5.1.1 Design of the BFM

In the SYSTEMC environment, a BFM is a hierarchical process that is derived from (using C++ inheritance) a SYSTEMC class called `sc_module`. The ports of this module correspond to the pins of the processor. The BFM class has several methods (which are member functions of C++ classes) that provide a programming interface to the software or to the ISS. The methods provided depend on the type of communication between hardware and software and are described below. The functionality of the BFM itself is modeled as a set of finite-state machines (that can execute in parallel).

In the SYSTEMC environment, the programming interface to BFMs is more or less fixed, i.e. the interface methods have the same prototype for all BFMs, though some BFMs may support methods that are not supported in others. This allows the user to swap one processor model for another easily, without having to change the C/C++ source code. This capability is important because it allows the user to explore different architectures with different processors. The various pre-defined types like `sc_address`, `sc_data`, `sc_register`, etc. (see below) can be specialized inside each BFM, therefore allowing complete freedom for each BFM to define these types appropriately.

5.1.2 Memory-mapped I/O

One of the most common architectures in systems-on-chips consists of CPUs and hardware devices connected to one or multiple memory buses. A portion of the address space is then allocated for each I/O device and hardware-software communication is implemented as memory accesses (memory-mapped I/O). A BFM provides the following methods to read and write memory:

```
void bfm_read_mem(sc_address addr, sc_data
*dat,int num_bytes)

void bfm_write_mem(sc_address addr, sc_data
dat, int num_bytes)
```

The `sc_address` and `sc_data` types are types defined in the BFM. For example, the address type can be an integer while the data type can be an array of bytes.

Software can access each device by performing memory reads and writes. When the software is implemented as a C/C++ program running on the host machine, explicit calls to the functions above can be added in the code to access I/O devices.

When an instruction set simulator is used, the software will make calls to the device drivers which will ultimately get converted to execution of memory read/write instructions. The ISS will then call the above methods in the BFM to perform the memory reads and writes. The BFM performs the appropriate bus transactions and, in the case of a memory read, sends the received data back to the caller of the method.

5.1.3 Interrupt-driven I/O

Interrupt signals may be used by external I/O devices and internal modules (timers, serial ports) to trigger an interrupt on the CPU. An interrupt controller, sensitive to the interrupt signals, is implemented as part of the BFM. Once the BFM detects an interrupt, it can execute a user defined function. The function to execute for a particular interrupt is specified using the following BFM method:

```
void bfm_register_handler (sc_interrupt intr,
void (*handler)(sc_interrupt))
```

The type `sc_interrupt` is defined in the BFM and contains information about the interrupt (the pin, the vector, etc). The `handler` function is provided by the user and when this function is invoked, it is provided with the details of the interrupt through its argument. Most types of interrupts are supported: synchronous, asynchronous as well as vectored interrupts. Besides, interrupts may be masked using the configuration ports.

When the software is implemented as a C/C++ program running on the host machine, the software writer has to use the method above to register interrupt handlers. When an instruction set simulator is used, the ISS software will register internal ISS routines as handlers for the interrupts.

5.1.4 Configuration ports, access to internal registers

CPU cores often have multiple modes of operation (e.g. little/big endian, multiple timer or serial modes, masked interrupts, etc.) according to the value set on the configuration ports and stored in internal configuration registers. Reconfiguration may be done by peripherals (through configuration ports) and by the software. The BFM provides the following method for accessing internal registers:

```
void bfm_write_reg(sc_register reg, sc_data
dat, int num_bytes)

void bfm_read_reg(sc_register reg, sc_data
*dat, int num_bytes)
```

The type `sc_register` is defined in the BFM. It contains all the information about a register (address, width, accessibility of each bit in the register, etc).

The support of configuration ports and registers involves the development of a controller in the BFM which maintains the value of the internal registers and other parameters. The value of these parameters and registers are then directly accessed by the other processes of the BFM. Note that the BFM usually models only the configuration registers and not the general purpose registers of the processor (though this can be added).

5.1.5 Timers and serial port

Some CPUs integrate serial I/O ports and timers. The timing of the transaction on these ports are controlled by timers internal to the CPUs. The controllers for the timers and the serial ports are implemented within the BFM. These timers and serial ports are accessed by writing special registers using the methods specified in the previous section. Note that the serial port controllers and the timers may send interrupts (e.g. to signal the end of a transaction, a real-time deadline or an error in the transmission).

5.1.6 Performance estimation functions

The BFM keeps track of the number of clock cycles used to perform various bus transactions and can provide a detailed report at the end of the simulation (or during each bus transaction). This reporting feature can be enabled by calling the following method:

```
void bfm_enable_tracing(int level)
```

Various levels of tracing is defined, whereby not only performance related information, but also debugging information can be provided by the BFM.

5.1.7 Hardware-software synchronization

In the actual hardware, the software and the hardware run in parallel. However during simulation, when a bus transaction is executed, the software is essentially stopped until the bus transaction is completed (i.e. by default, the BFM read and write functions are blocking). This essentially serializes the execution of the hardware and software. Designers have the choice of making the software execute in parallel with the hardware, by setting a flag in the BFM (a function is provided for this). When the flag is set, the BFM routines return instantaneously, allowing the software execution to go ahead.

When using a BFM stand-alone, explicit timing information may also be added in the software through the use of `wait()` or of the following method call:

```
void bfm_idle_cycle(int cycles)
```

These calls suspend the execution of the software for one clock cycle or more. Nevertheless, this technique has limitation for modeling the timing of complex superscalar pipelined architectures. Therefore, for timing accurate simulation, a BFM needs to be used with an ISS.

5.2 Instruction set simulator

For a given architecture, an instruction set simulator (ISS) reads the assembly code written for the architecture and simulates it on a host machine. Different types of ISS can be developed for different purposes.

First, the verification of the functional correctness of an application written in assembly code can be performed using ISS. For this purpose, very fast ISS can be developed by translating the instructions of the target architecture into instructions on the host machine. This technique has been applied in [16] for hardware/software co-verification. However, this type of ISS cannot be cycle-accurate for complex pipelined, superscalar architectures.

Second, the verification of the timing and interfaces (buffer size, bus contention) between the different components of the system can be done using an ISS and a BFM. Here, the timing accuracy is usually important and the ISS is often implemented as an emulator.

Our environment makes no assumptions about the nature of the ISS that the designer chooses to use, as long as the ISS can be integrated with BFM. However, if the designer requires performance estimates, we recommend that the designer chooses an ISS that can provide accurate cycle counts. Also if the designer wants certain types of simulation speed improvements, then the ISS is required to provide the necessary 'hooks' for that (these 'hooks' are described in Section 5.4).

5.3 Integrating an ISS and a BFM

By integrating an ISS and a BFM a complete processor model can be generated and used for detailed simulation. If the ISS is cycle accurate, then the entire processor model is cycle accurate (note that a BFM is a cycle accurate model to begin with).

An ISS typically consists of fetch, decode, and execute units. The fetch unit reads the current instruction in the instruction memory or in a file. The decode unit decodes the instructions and outputs the opcode as well as the different operands. Finally the execute unit performs the operation, reads/writes memory, updates registers and computes the address of the next instruction. A typical ISS consists of these three units. In the execution unit of the ISS, calls are made to the BFM interface methods to execute memory and configuration register reads/writes. The fetch unit of the ISS also makes call to BFM methods to get instructions and data.

For more complex architectures (e.g. superscalar, pipelined) several units must be added to implement pre-fetch, issue and write-back stages. Level 1 cache would also be modeled as part of the ISS. Moreover, the interrupt controller used in the BFM should also be modified to take into account some of the features of modern processors (pipeline, reorder buffer, etc.). For such complex architectures, various parts of the ISS may need to call BFM methods.

In addition to this, the ISS may need to provide the BFM with certain memory access functions. These functions are described in Section 5.4.

5.4 Supporting techniques for speeding up simulation

Several techniques have been presented in [4,9,16] that improve simulation speed by reducing the amount of activity that needs to be simulated. Our environment supports these techniques easily and efficiently.

5.4.1 Reduce activity on memory bus

For most applications more than 95% of the traffic on the memory bus can be attributed to instruction and data fetches. If the functionality of the processor bus interface with the instruction and data memory has been verified, then there is no need to simulate this activity during co-simulation.

The simulation of instruction fetch can be avoided by putting the instruction memory as part of the ISS. The ISS is modified to access instruction memory directly. Such a technique can also be applied to the data memory, thereby eliminating data access simulation. However, since external devices may need to access the data memory, the BFM can be configured to recognize bus cycles where an external device is accessing data memory. The BFM can then modify the data memory directly inside the ISS. For this, the ISS needs to provide a set of functions that will be used by the BFM to read and write the data memory. The ISS also needs to implement a

memory map whereby addresses in certain ranges will be accessed directly while addresses in other ranges will cause the bus cycles.

Note that the same technique can also be applied to any other device on the memory bus with little additional re-coding.

5.4.2 Turn off clocks on modules

All processes connected to the memory bus receive the clock signal, irrespective of whether the process is addressed by the CPU or not. Since the clock signal forces evaluation of at least a part of the process functionality of each process, it is wasteful. By turning off the clock for all instances except when the processor addresses devices on the bus, we can speed up simulation. This is supported in our framework by having the BFM generate the bus clock for the system. The BFM will generate the bus clock only when devices on the bus are addressed.

6. IMPLEMENTATION: 8051

In this section we present our model of BFM and cycle-accurate ISS for the Synopsys DesignWare 8051 core [12]. The dw8051 macrocell is a configurable, fully-synthesizable, and reusable 8051 core. It is fully binary compatible with the industry standard 803x/805x microcontrollers.

The 8051 is an 8 bit microcontroller widely used in simple embedded application such as in smartcard, cars, toys, etc. It supports many of the I/O modes found in other processors. A block diagram of a system based on the dw8051 core is presented in Figure 3. Hardware devices can be connected to the memory bus or to the Specific Function Register (SFR) bus. The dw8051 also features six interrupt ports (extendable to twelve), up to two serial ports and one or two timers. The timers and serial ports have respectively three and four different modes of operation which makes them fairly complex to model. For more details on the dw8051 architecture, refer to [12].

6.1 dw8051 bus functional model

We developed a dw8051 BFM using the SYSTEMC framework. In particular, our BFM supports:

- timer 1, mode 0,1,2
- serial port 0, mode 0,1,2,3
- external interrupts
- external memory accesses with variable stretch cycles
- SFR (Specific Function Register) accesses

A system consisting of the dw8051 BFM and the testbench is depicted in Figure 4. The BFM (`8051BFM` process) consists of several processes. The software interface process (`sw_interface`)

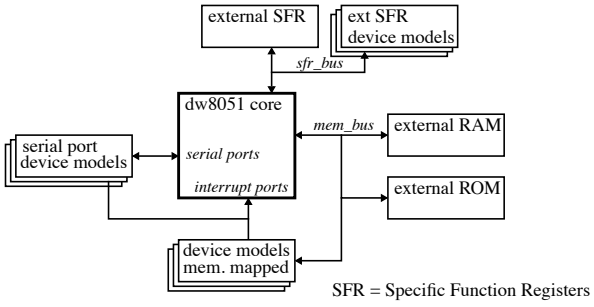

Figure 3: block diagram of the DW8051 core

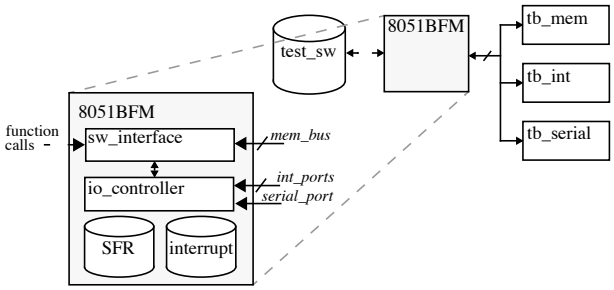

Figure 4: co-verification with dw8051 bus functional model

provides the BFM programming interface functions. The `io_controller` process updates the values of internal specific function registers (SFR) and models the behavior of the timer and serial port. Both processes access methods of the `interrupt` controller, implemented as a separate class.

The dw8051 BFM has been tested with an example system. The software part (`test_sw`) of this example consists of several test-cases to interactively test the different features of the BFM. In addition to the BFM, the hardware part consists of several processes (`tb_mem`, `tb_int`, `tb_serial`) modeling the behavior of devices on the memory bus and on the serial port.

File	Line of C++
BFM	1944
HW testbench	497
SW testbench	1134

Table 1: number of lines for BFM and testbench

Implementation	Testbench			
	Memory	SFRs	Serial	Timer
SYSTEMC	383	368	635	929
Co-simulation	684	460	588	642

Table 2: simulation time (in seconds) for the dw8051 system

The number of lines for each process is presented in Table 1. The results in Table 2 compare our methodology to a traditional HDL-based co-simulation methodology. The first row shows the simulation time for the system depicted in Figure 4, where the entire system is modeled in the SYSTEMC environment. The software/hardware test programs are testing the different features of the BFM (memory port, sfr update, serial ports and timers). The simulations have been measured on Sun Ultra-Enterprise server with eight 336MHz UltraSparc II processors and 2GB RAM.

The second row shows simulation time for the same system, where the software runs on a workstation and communicates through some form of IPC (sockets) to the hardware. This is the way co-simulation is done in all HDL-based commercial tools. Note that only the overhead due to the use of IPC is taken into account (i.e. the hardware is still described in C/C++). When the hardware is described using an HDL, one would also typically have to add the effect of possibly less efficient HDL models and the overhead of the HDL simulator itself. As can be seen, our environment provides a speed-up of as much as 44% due to the simplification of the communication between hardware and software (cf. test of the memory and SFRs). However, the traditional approach may exploit more of the parallelism between the SW and HW on distrib-

uted systems, improving the results when the communication between HW and SW is reduced (cf. test of serial port and timer).

6.2 dw8051 cycle-accurate model

A cycle-accurate ISS for a subset of the dw8051 architecture has also been developed and integrated with the BFM described in the previous section. The ISS is implemented as a single process that fetches, decodes and executes the 8051 instructions.

For this test, an I/O device has been connected on the memory bus. This generic device has a latency of 100 clock cycles. The first row shows the simulation time without any optimization. The second row presents the results after applying the techniques described in Section 5.4. These optimizations provides a 94% speed-up. Finally the last row presents the results when the software is running directly on the host machine. In this example, the ISS is quite simple. Therefore, the overhead for using an ISS is only about 10%. It would typically increase for more complex architecture (e.g. superscalar pipelined architectures).

Implementation	Simulation time
ISS + BFM	4,708
optimized ISS + BFM	279
C/C++ + BFM	252

Table 3: simulation time for the cycle accurate model (in s)

7. CONCLUSION

We have presented our environment for hardware-software co-verification in C/C++ using SYSTEMC. This approach to choosing an environment for co-simulation and to creating models has several advantages.

Our environment is based on the use of a single language and a uniform modeling paradigm for both synthesis and simulation. The interaction between the software part, running directly on the host machine or emulated by the ISS, and the hardware is therefore simplified. We don't need any of the complex co-simulation mechanisms to interface HDL simulators with the software world.

Our environment allows designers to model their system entirely using C/C++. By design, the C/C++ language can be very efficiently compiled on today's architectures, enabling the development of very fast models. In addition, there are no overheads associated with interfacing a C world with an HDL world. Moreover, our environment also offers flexibility and therefore supports most of the techniques used to speed-up hardware software co-simulation [4,9,16].

By proper design of the BFM and ISS, performance estimates can be obtained for software execution, which can be used to drive the codesign process. Having software and hardware described in the same language together with a good object oriented (OO) design technique will make moving functionality between software and hardware easier.

Finally, C++ allows us to use OO techniques to create BFMs and ISSs that can be reused from one generation of processors to the next, making the job of developing and maintaining these models simpler.

8. REFERENCES

[1] David Becker, Raj Singh, Stephen Tell, "*An environment for hardware/software co-simulation*", proceedings of the 29th design automation conference, pp. 129-134, 1992.

[2] CoWare N2C, http://www.coware.com

[3] Giovanni De Micheli, "*Hardware Synthesis from C/C++ Models*", proceedings of the DATE'99 conference, pp.382-383, March 99.

[4] Abhijit Ghosh et al. "*A hardware-software co-simulator for embedded system design and debugging*", proceedings of the ASP-DAC'95, pp. 155-164, 1995.

[5] Abhijit Ghosh, Joachim Kunkel, Stan Liao, *"Hardware Synthesis from C/C++,"* proceedings of the DATE'99 conference, pp.382-383, March 99.

[6] Rajesh Gupta, Claudionor Coelho, Giovanni De Micheli, "*Synthesis and simulation of digital systems containing interacting hardware and software components*", proceedings of the 19th Design Automation Conference, pp. 225-230, 1992.

[7] Michael Keating, Pierre Bricaud, "*Reuse Methodology Manual For System-on-Chip Designs*", Kluwer Academic Publishers, 1998.

[8] Stan Liao, Steve Tjiang, Rajesh Gupta, *"An Efficient Implementation of Reactivity for Modeling Hardware in the SCENIC Design Environment,"* proceedings of the Design Automation Conference DAC'97, pp.70-75, June 97.

[9] Mentor Graphics Seamless, http://www.mentorgraphics.com/codesign/main-f/index.htm

[10] Claudio Passerone, Luciano Lavagno, Massimiliano Chiodo, Alberto Sangiovanni-Vintentelli, "*Fast hardware/software co-simulation for virtual prototyping and trade-off analysis*", Proceedings of the Design Automation Conference DAC97, pp. 389-394, Anaheim 1997.

[11] James Rowson, "*Hardware/Software Co-simulation*", proceedings of the 32nd Design Automation Conference, pp. 439-440, June 94.

[12] Synopsys DesignWare DW8051 MacroCell http://www.synopsys.com/products/designware/8051_ds.html

[13] Synopsys Eagle http://www.synopsys.com/products/hwsw/hwsw.html

[14] SystemC http://www.SystemC.org

[15] C. A. Valderrama et al. "*A unified model for co-simulation and co-synthesis of mixed hardware/software systems*", proceedings of the European Test Conference, pp. 180-184, 1995.

[16] Vojin Zivojnovic, Heinrich Meyr, "*compiled HW/SW co-simulation*", proceeding of the 33rd Design Automation Conference, pp. 690-695, Las Vegas, June 96.