Component-Based Hardware/Software Co-Simulation [*]

Ping Hang Cheung
Dept. of ECE
Portland State University
Portland, OR 97207, USA
cheung@ece.pdx.edu

Kecheng Hao
Dept. of Computer Science
Portland State University
Portland, OR 97207, USA
kecheng@cs.pdx.edu

Fei Xie
Dept. of Computer Science
Portland State University
Portland, OR 97207, USA
xie@cs.pdx.edu

Abstract

Developing highly efficient and reliable embedded systems demands hardware/software (HW/SW) co-design and, therefore, co-simulation. In order to be highly configurable, embedded systems are increasingly component-based in both hardware and software. In this paper, we present a novel approach to hardware/software co-simulation of component-based embedded systems. Our approach features a component model which unifies hardware and software component models with the concept of bridge component. Bridge components raise the level of abstraction for designing HW/SW interfaces. Their specifications are used to configure the co-simulators. Our approach has been applied to co-simulation of sensor system instances included in the TinyOS distribution. The case studies have demonstrated that our approach is readily applicable to real-world embedded systems and reduces co-simulation complexities.

1 Introduction

In this paper, we present a novel approach to hardware/software (HW/SW) co-simulation of component-based embedded systems. Our approach is based on a component model that unifies hardware and software component models. The key to this component model is the concept of bridge component, which raises the level of abstraction for designing HW/SW interfaces. Bridge components inter-connect hardware and software components and propagate events across the HW/SW boundaries. The abstraction using bridge components is enabled by a platform-specific bridge specification language and its supporting compiler. For co-simulation, this compiler compiles the bridge components into hardware and software executables and necessary configurations for co-simulators.

We have realized our approach in development of a co-simulation utility for sensor systems based on TinyOS [6]. This utility supports software components written in the nesC [4] programming language and hardware components written in Verilog, VHDL, or SystemC. We have applied this utility to the sensor system instances in the TinyOS distribution. In these applications, we re-engineered the hardware components based on our understanding of the Motes [6] platforms. The case studies have shown that our approach is very effective in enabling co-simulation of embedded systems and reduces co-simulation complexities.

The reminder of this paper is organized as follows. In Section 2, we provide the background of this work. In Section 3, we introduce the unified component model. In Section 4, we present our approach to component-based co-simulation. In Section 5, we discuss related work. In Section 6, we conclude this paper and present future work.

[*]This research was supported by Semiconductor Research Corporation, Contract RID 1356.001.

2 Background

In this section, we discuss two simulators, Giano [3] and ModelSim [9], upon which we realize our approach.

2.1 Giano System Simulator

Giano is a full-system simulator for embedded systems. It incorporates simulation of processors and peripherals of an embedded system. A hardware simulator can be attached to Giano through a Programming Language Interface (PLI) [8] and be responsible for simulation of hardware designs written in HDLs such as Verilog, VHDL, and SystemC. Giano is also capable of simulating hardware designs written in C.

2.2 ModelSim Hardware Simulator

ModelSim is a hardware simulator which is capable of simulating hardware designs written in HDLs such as Verilog, VHDL, and SystemC. It supports hardware debugging by providing signal traces, waveform analysis, code coverage measurement, etc. It also supports Assertion-Based Verification (ABV) using the IEEE Property Specification Language (PSL) [7]. In addition, it supports communication with other software systems through PLIs.

3 Unified Component Model

A key challenge in co-design and co-simulation is the HW/SW semantic gap. In the common co-design practice, the HW/SW interfaces are often designed at a low abstraction level in terms of variables, function calls, and signals. The semantic entities in the interfaces are scattered in hardware and software. This compromises the component-based structures of both hardware and software.

3.1 Unified Component Representation

We have developed a unified component model for hardware and software, which raises the level of abstraction for HW/SW interface specification via the concept of bridge component. Bridge components inter-connect hardware and software components and propagate events across HW/SW boundaries. Bridge components abstract an embedded system platform including its processor, memory, bus, and operating system and provide an abstract and concise way to specify HW/SW interfaces.

In our unified component model, a component has three elements: executable, interface, and properties. The executable is the implementation of the component. In the realization of the component model for sensor systems, for a software component, its executable is specified in nesC; for a hardware component, its executable is specified in a HDL such as Verilog and VHDL; for a bridge component, its executable is specified in a bridge specification language (See Section 3.2). The interface of a component defines the semantic constructs for interacting with other components. The interface of a software or hardware component is determined by its corresponding design/implementation language. For a bridge component, it has both a hardware interface and a software interface, for interacting with hardware and software. Properties are temporal assertions over the behaviors of the component. Evaluation of temporal assertions over the behaviors of an entire system requires verification (e.g., simulation) across HW/SW boundaries. (A system is also a composite component.) Evaluation of assertions is bounded by the software and hardware domains.

Figure 1 illustrates a sensor system structured following the unified component model. The software components are developed in nesC and their interfaces consist of commands and events, which are essentially C functions. Figure 2 illustrates the interfaces of the software components. The hardware components are developed in Verilog and their interfaces consist of signals. Figure 3 illustrates the interface specification of the hardware components. The output of the hardware sensor component contains a data path and an

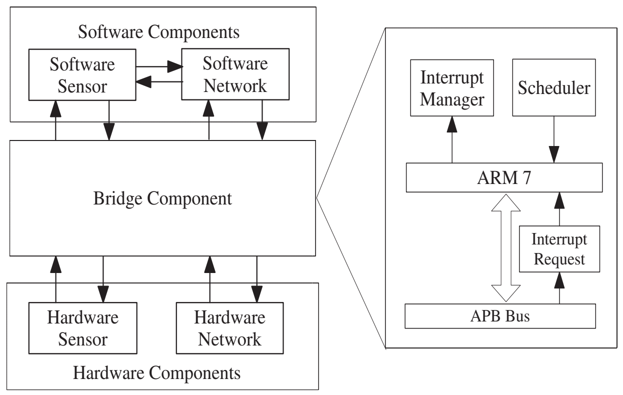

Figure 1. Sensor System Architecture

```
Interface StdControl{
  command init(); command start(); command stop();
}
Interface DataTran{
  command SendData(int data); event ACK();
}
Interface Interrupt{
  event IntrSensor(); event IntrNetwork();
}
```

Figure 2. Example Software Interfaces

interrupt and the input is a set of control signals indirectly from the software components via the bridge component. (The designs/implementations of the hardware and software components are not shown due to space limitation.)

3.2 Bridge Components

A bridge component is an intermediate component that inter-connects hardware and software components. It has dual interfaces, a hardware interface and a software interface, for instance, the bridge component shown in Figure 1. The hardware (or software, respectively) interface of this component contains the same signals (or functions) from the interfaces of the hardware (or software) components while reversing their input and output directions.

The executable of a bridge component is specified in a bridge specification language (BSL). BSL provides an abstract way for designers to specify hardware and software interactions through defining the mappings between hardware and software events. These mappings are automatically compiled or synthesized into software or hardware implementation for the final system implementation and used to configure the co-simulator automatically. BSL is platform-specific in that how the mappings are specified depends on the hardware and software design/implementation languages for an embedded system platform and the implementation of the mappings depends on the processor model, the bus model, the OS, etc. For a widely used platform, it

```
Sensor P1(                      Network P2(
   .PCLK(PCLK),                    .PCLK(PCLK),
   .PSEL(PSEL1),                   .PSEL(PSEL2),
   .PENABLE(PENABLE),              .PENABLE(PENABLE),
   .PRESETn(PRESETn),              .PRESETn(PRESETn),
   .PADDR(PADDR),                  .PADDR(PADDR),
   .PWRITE(PWRITE),                .PWRITE(PWRITE),
   .PWDATA(PWDATA),                .PWDATA(PWDATA),
   .PRDATA(PRDATA),                .PRDATA(PRDATA),
   .INTR(INTR1)                    .INTR(INTR2),
);                                 );
```

Figure 3. Example Hardware Interfaces

is justified to develop a BSL and its supporting compiler, in order to simplify system design and verification.

We have developed a BSL for the sensor system platform and its supporting compiler. Figure 4 shows example hardware/software event mappings specified in this BSL. The left operands are software functions where the right

```
Mapping{
   SensorM.SensorData() ⇐ Sensor.PRDATA;
   NetworkM.Network() ⇒ Network.PWDATA;
   SensorM.IntrSensor() ← Sensor.INTR1;
}
```

Figure 4. Example Mappings

operands are the hardware signals. The operator ⇐ maps hardware signal *Sensor.PRDATA* to the software function *SensorM.SensorData()*. This operator assigns the data value from the right operand (HW) to the left operand (SW). The right operand *PRDATA* is an output of a 32-bit bitvector in the hardware *Sensor* module where the left operand *SensorData()* is an input function in the software *SensorM* component whose return value is a unsigned 32-bit integer. The function *SensorData()* will return the value of *PRDATA* to its caller in the software domain. The operator ⇒ maps software function *NetworkM.Network()* to the hardware signal *Network.PWDATA*. The parameter of the function *Network()* will be written to the signal *PWDATA*. The ← operator is used to map a hardware interrupt to a software interrupt function. For instance, the hardware interrupt signal *Sensor.INTR1* is mapped to the interrupt function *SensorM.IntrSensor()* in Figure 4. The BSL compiler processes these mappings and generates configurations for Giano and ModelSim to configure the PLI for HW/SW interactions.

4 Component-Based Co-Simulation

Our approach to component-based co-simulation has three major features: (1) platform-oriented co-simulator configuration; (2) simulator synchronization based on bridge components; (3) temporal assertion evaluation in co-simulation. We will elaborate on these features and illustrate them with a co-simulator for component-based sensor systems.

4.1 Platform-Oriented Co-Simulator Configuration

For different embedded system platforms, the ways how the co-simulators are configured differ. However, given a platform, there is much commonality in configuring the co-simulator for different systems based on this platform. The bridge components specified in BSL abstract the core platform components of an embedded system, such as its processor, bus, scheduler, and interrupt manager. Therefore, these components are invisible but configurable via BSL. This abstraction is only possible with compiler support, in particular, the platform-specific BSL compiler.

Figure 5 illustrates the architecture of a co-simulator for the sensor systems introduced in Section 3. The keys of

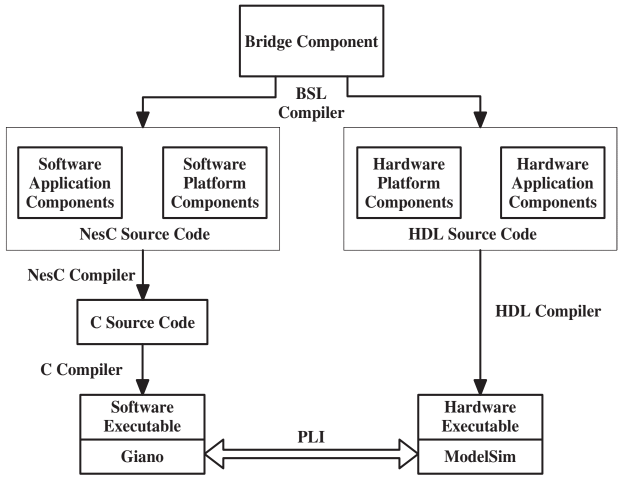

Figure 5. Co-Simulation Architecture

this architecture are the compilers for hardware, software, and bridge components. The BSL compiler generates the hardware platform components in Verilog such as the bus and the interrupt queue and the software platform components in nesC such as the interrupt manager, the scheduler, the main loop of the program, etc. Furthermore, the BSL compiler configures Giano and ModelSim with the platform components of an embedded system such as the processor. It is also responsible for establishing *mappings* between hardware signals and software functions/variables by generating hardware code in Verilog and software code in nesC. The HW/SW event mappings are also used to configure the PLI between Giano and ModelSim, which is used for HW/SW communication and synchronization. The software (or hardware, respectively) components including those generated by the BSL compiler are compiled into executables by the software (or hardware) compiler.

4.2 Synchronization Based on Bridge Components

The synchronization between hardware and software components is done through the PLI between Giano and Model-

Sim. The PLI is masked by bridge components. Hardware and software components interact via bridge components whose semantics dictate the synchronization scheme. This scheme is realized in the co-simulation through the BSL compiler. In our networked sensor system example, a simple synchronization scheme is supported. The scheme includes a prioritized hardware request queue and a software scheduler. When the hardware requests attention from the software. It triggers a hardware interrupt and pushes it into the hardware request queue. When the software picks up the interrupt request, it carries out the pre-defined task in response to the interrupt. Once the transaction completes, the next event on the hardware queue is executed. When the queue is empty, other software tasks can be scheduled by the scheduler. In essence, the HW/SW synchronization is event-driven and a transactional semantics is enforced.

4.3 Assertion Evaluation in Simulation

We have extended PSL into an assertion language over both hardware and software, namely xPSL [12], and developed an xPSL assertion evaluation utility for co-simulation, shown in Figure 6. The utility includes a software event monitor, a hardware event monitor, and an assertion evaluation engine. The software event monitor (or hardware event monitor, respectively) observes the occurrences of software events (or hardware events) and notifies the xPSL assertion evaluation engine. The xPSL assertion engine processes the events and determines whether any assertion is violated.

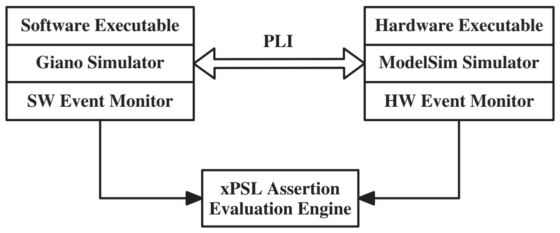

Figure 6. Assertion Evaluation Utility

4.4 Preliminary Experiments

We have applied our approach to the sensor system instances included the TinyOS distribution. For each system, we utilized their software components with minor modifications to the lowest level components, we re-engineered their hardware components following the unified component model, and we also introduced the bridge components. In addition, we also introduced xPSL assertions over the HW/SW interactions and evaluated the assertions using component-based co-simulation. The case studies show that our approach reduces co-simulation complexities through the abstraction based on bridge components and is very effective in evaluating assertions across HW/SW boundaries.

5 Related Work

There has been much research [1, 5, 11, 2] on co-simulation that led to industrial tools such as Mentor Graphics Seamless [10] and Microsoft Giano [3]. In [1], a co-simulation environment was developed based on the Verilog PLI. In [5], the co-simulation and co-synthesis of hardware and software components were explored. In [11], embedded systems were co-simulated and co-verified as pure software specified in C/C++. In [2], modeling, simulation, and design of concurrent real-time embedded systems was studied, with a focus on assembly of concurrent components. Our work builds on these previous work, provides efficient co-simulation support to component-based embedded systems and simplifies configuration of co-simulators.

6 Conclusions and Future Work

In this paper, we have presented a component-based approach to HW/SW co-simulation, which allows hardware and software to be simulated in a unified environment while maintaining modularity of hardware and software components. Case studies on sensor systems have shown that our approach is readily applicable to real-world embedded systems and reduces co-simulation complexities. Our future work will be focused on integration of co-simulation and formal co-verification based the unified component model.

References

[1] D. Becker, R. K. Singh, and S. G. Tell. An engineering environment for hardware/software co-simulation. In *DAC*, 1992.

[2] Berkeley. Ptolemy project. http://ptolemy.eecs.berkeley.edu/index.htm.

[3] A. Forin, B. Neekzad, and N. L. Lynch. Giano: The two-headed system simulator. Technical Report MSR-TR-2006-130, Microsoft Research, 2006.

[4] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems,. In *Proc. of PLDI*, 2003.

[5] R. Gupta, C. Coelho, and G. D. Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *DAC*, pages 225–230.

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Proc. of ASPLOS*, 2000.

[7] IEEE. *IEEE Standard for Property Specification Language (PSL) (IEEE Std 1850-2005)*. IEEE, 2005.

[8] IEEE. *IEEE Standard for Verilog (IEEE Std 1364-2005)*. IEEE, 2005.

[9] Mentor Graphics. ModelSim. http://www.mentor.com.

[10] Mentor Graphics. Seamless. http://www.mentor.com.

[11] L. Séméria and A. Ghosh. Methodology for hardware/software co-verification in c/c++. In *ASP-DAC*, 2000.

[12] F. Xie and H. Liu. Unified property specification for hardware/software co-verification. In *Proc. of COMPSAC*, 2007.