Proceedings of the 2018 Winter Simulation Conference
M. Rabe, A.A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, eds.

CO-SIMULATION OF HARDWARE RTL AND SOFTWARE SYSTEM USING FMI

Masudul H. Quraishi
Hessam S. Sarjoughian
Soroosh Gholami

Arizona Center for Integrative Modeling and Simulation
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, AZ 85281, USA

ABSTRACT

Software-hardware co-design enabled with co-simulation is useful for building embedded computing systems. Indispensable to design is developing hardware and software simulation models at appropriate abstraction levels. Toward this goal, this paper presents a study of combined Register-Transfer-Level (RTL) and software system modeling. Specifically, composition of hardware and software models is proposed and a co-simulation environment to simulate the models is developed. The hardware and software parts of a prototypical Network on Chip (NoC) system are modeled and simulated. The hardware part is specified at RTL level using the DEVS Suite Simulator and the software part defined as a MATLAB script. A Functional Mock-up Interface (FMI) is developed for the DEVS-Suite Simulator to support hardware and software model coupling and co-simulation. This study details a modular development of hardware and software models executing on disparate environments instead of employing a monolithic modeling method supported with a monolithic simulation engine.

1 INTRODUCTION

Abstraction taxonomies are known to be useful in distinguishing models intended for different, and sometime complementary, purposes. A taxonomy provides concepts and definitions to tackle challenges inherent in capturing model complexity. Resolutions for time, data, function, and structure are among key contributors, for example, to the development of Systems-on-Chips (SoC) and more broadly for Internet of Things (IoT). Digital systems, as well as mixed-signal systems, are commonly developed according to the Hardware Description Language (HDL) (Shiva 1979). One well-known and successful realization for the HDL is the Very Hardware Description Language (VHDL) (Standard 1988). It supports modeling at behavioral and RTL abstraction levels (Mano et al. 2015). A behavioral model abstraction can be defined as functions and timing using assignment expressions and timed communication signals. A Register Transfer Level (RTL) model defines structure and behavior in terms of combinational and sequential logic gates under the control of a clock and data signals. These models are intended to be agnostic to a myriad of FPGA (Field Programmable Gate Array) implementations. An important factor in the widespread use of VHDL is the availability of tools like Quartus (Intel FPGA 2018) that can support transforming behavioral to RTL to Logic abstraction levels as well as automation for FPGA code generation.

As the role of software continues to grow and increase system complexity, it is attractive to use discrete event modeling methods. For example, to model Network-on-Chip systems, it is important to have models that are accurate in terms of timing for asynchronous events and efficient execution (Ghosh 2001). A candidate modeling approach is the Discrete Event System Specification (DEVS) (Zeigler et al. 2000) formalism. Together the IO System and Coupled DEVS models can be viewed to be at a similar abstraction level as the behavioral VHDL model.

Given the system-theoretic foundation of DEVS, it can be specialized at the RTL abstraction level (Chen and Sarjoughian 2009). A library of combinatorial and sequential DEVS RTL model elements were developed for MIPS32 and implemented in the DEVS Suite Simulator (ACIMS 2018). Use of the DEVS modeling framework encourages formal model specifications. Recently, methods and tools are developed to bridge the gap from mathematical specification to code with the aid of Unified Modeling Language (UML), Statecharts (Fard and Sarjoughian 2015) and UML Activity Diagrams (Sarjoughian et al. 2015). These modeling capabilities afford behavioral modeling that are beneficial to DEVS. RTL model can be developed more rigorously and systematically based on the formal DEVS specification with the semi-formal Statecharts and UML Activity methods.

Designs for simulation and model checking of embedded systems benefit from a methodology where software and hardware components at different abstraction levels can be validated and verified (Gholami and Sarjoughian 2017). For such systems, it is important to use co-simulation to reduce design complexity and thus verification and validation efforts. For co-simulation, tools using their own simulation protocols (e.g., solvers) allow combining multiple domains with different time steps. Functional Mock-up interface (FMI) (Blochwitz et al. 2011) is a standard framework to couple simulators. In conventional FPGA design flow, hardware and software components are designed and validated separately. Using FMI allows to having a co-simulation environment where mixed hardware RTL and software models can be simulated separately. Co-simulation naturally lends itself to distributed execution since models can be executed as black-boxes relative to one another. To demonstrate this co-simulation, we propose - integrating the DEVS-Suite Simulator with MATLAB using the FMI standard. We have developed a co-simulation environment for hardware (RTL) DEVS models and software MATLAB models. As exemplar NoC component, we modeled a buffer for the incoming flits. In the presence of virtual channels, one of the most elegant methods of buffer design is circular (Ni et al. 1998). We consider a circular buffer with four cells, two virtual channels, and an on/off flow control mechanism. We developed this component in DEVS at RTL abstraction level. We extended this RTL design with a router modeled as a microprocessor and a firmware embedded in it. The firmware is abstracted as a MATLAB script which will be referred to as software model in the remainder of this paper. MATLAB is used to demonstrate the developed FMI for the DEVS-Suite simulator and using it for DEVS and non-DEVS simulations to interoperate.

The contribution of this paper is as follows. A generic DEVS-FMI interface is developed for the DEVS Suite Simulator. Basic NoC models are developed using DEVS and MATLAB. Then, the developed interface is used to achieve co-simulation of the RTL and software model. This kind of co-simulation is valuable as it allows independently developed software models as part of designing embedded software systems. The paper also discusses research on DEVS-based High Level Synthesis (HLS).

2 BACKGROUND

This section briefly describes Network-on-Chip (NoC) system, its model development and simulation in DEVS, and the Functional Mock-up Interface (FMI).

2.1 Network-on-Chip (NoC)

DEVS is grounded in systems theory which provides modeling using a formal specification. DEVS models are realized using an object-oriented programming language (Java) in the DEVS-Suite Simulator. DEVS employs the foundational concepts of components with I/O ports, modularity, and hierarchy. The I/O ports can be designed to have arbitrary values and they can be restricted. The dynamics of an atomic model in DEVS are defined in terms of external, internal, output, and time-advance functions. The first two define state transitions due to receiving inputs as external events and internal state changes. The latter two define generation of output events and time duration for each of the states the model can be in. Two primary state variables, phase and sigma, store the phase and timing information of the model. Other state variables (referred to as secondary) are needed to specify states of target system being modeled.

A Network-on-Chip (NoC) is a communication system (Dally and Towles 2004) proposed to manage the interactions between Intellectual Properties in a System-on-a-Chip (SoC). The regular structure, cost-effectiveness, and high bandwidth are the main reasons NoCs for large SoCs. In a NoC, streams of data are packetized and sent over the network. Routers are in charge of directing this packetized data (flits) toward their destinations. Flits go through 6 major phases in a router: queueing, routing, virtual channel allocation, switch allocation, switch traversal, and transmission. In this paper, we focus on the queueing phase where flits are queued in a buffer until they are dequeued under an overflow control mechanism.

2.2 Functional Mock-up Interface (FMI)

The Functional Mock-up Interface (FMI) is proposed as an interface for model exchange and co-simulation of dynamic models (Blochwitz et al. 2011). This interface is devised to support model exchange and co-simulation. A component which implements the FMI interface is known as Functional Mock-up Unit (FMU). As a package, it contains model functionality as C code and information about the model as XML document. For model exchange, one modeling environment exports a model as FMU. The FMU can be imported and used by another FMI-compliant modeling environment. With FMI two or more simulation environments can be coupled to form a co-simulation environment. One of the simulation environment must be master while all others are slaves. The master simulator is responsible for synchronized data exchanges and executions of all slave simulators relative to its own execution cycles. An FMI adapter needs to be developed for slave simulator to communicate with the master. The adapter is a dedicated simulation component responsible for information retrieval from FMU and data exchange between master and slave simulators. In this paper, a C++ wrapper library for FMI 1.0 specification is used, which is called FMI++. In subsequent sections, FMI and FMI++ are used interchangeably.

3 RELATED WORK

Development of efficient and reliable embedded systems requires hardware/software co-design and co-simulation. Numerous co-simulation environments and frameworks are introduced to tackle design and validation complexity of systems. Some of these related works are described in this section.

3.1 FMI based Co-simulation

MECSYCO is a middleware developed for co-simulation of cyber physical systems (Camus et al. 2018). Co-simulation engine of MECSYCO utilizes DEVS formalism to integrate tools that use different formalisms, allowing homogeneous co-simulation of heterogeneous tools. A DEVS wrapping strategy is proposed for the FMU component in order to integrate varieties of continuous modeling and simulation (M&S) tools in MECSYCO. While MECSYCO integrates simulation tools under the same formalism using DEVS wrappers, our approach connects simulation tools differently. We have created a **DEVS-FMI interface** to connect simulation tools with DEVS-Suite Simulator using FMI as middleware. In MECSYCO, if an M&S tool uses a formalism that does not comply with DEVS, it cannot be integrated. In our approach, compliance with DEVS formalism is not needed. However, an M&S tool must provide an FMI adapter to connect.

A co-simulation of a distributed application with FMU using SimGrid is proposed (Camus et al. 2016). In this work, co-simulation of equation-based FMU with a code-based distributed application is presented by embedding FMU generated from one simulation tool (OpenModelica) into another tool (SimGrid) as a dedicated model. Unlike our work, the co-simulation is achieved through the exchange of model I/O rather than using the concept of interoperability of simulation tools. This represents a weak form of co-simulation. In our work, the FMU generated from MATLAB is not embedded into DEVS-Suite Simulator, it is used to connect with MATLAB using interoperability with FMI.

3.2 Co-simulation involving Hardware Models and Simulators

A High-Level Architecture (HLA) based co-simulation framework for mixed signal SoC design is proposed (Seok et al. 2017). The co-simulation framework provides a formal interface to integrate new simulation tools and connect with existing hardware simulators and emulators in the framework. The framework utilizes HLA-based interoperability between mixed signal heterogeneous hardware models and tools. The simulation tool to be integrated into the framework needs a Signal Event (SE) converter. Compared to this work, our models are abstracted at RTL level and we use FMI based interoperability to connect simulation tools. The simulation tools use FMI adapter to connect to DEVS-FMI co-simulation environment.

SystemC (SystemC 1999) is a C++ library supporting transaction level modeling (TLM) (Maillet-Contoz and Strassen 2005) as an intermediate abstraction between Register Transfer Level (RTL) and architecture-independent abstractions. TLM allows specifying systems by excluding details of communication architecture. SystemC is designed to support modeling, simulation, and architectural exploration of TLM models. The modeling process is grounded on classes and macros developed in C++. One important advantage of SystemC (in relation to Hardware Description Languages) is its support for modeling both hardware components and embedded software. This enables designers to explore partitioning of system functionality into hardware and software components. SystemC does not have an RTL model library. However, tools like Vivado HLS (Vivado Design Suite 2018) provide an automatic transformation from SystemC specification to RTL as well as co-simulation of SystemC and RTL. A co-simulation of SystemC TLM and RTL is proposed by (Park et al. 2008) for system verification. Compared to SystemC, DEVS is not restricted to TLM only. Moreover, DEVS has a library of combinational and sequential RTL models (Chen and Sarjoughian 2009). We used these models to design circular buffer at RTL level.

HDL Verifier tool (MathWorks 2017) of MATLAB & Simulink allows linking HDL simulator with MATLAB and Simulink in a client server configuration. This approach is analogous to frontend backend approach used for simulation tool coupling (Widl and Müller 2017). MATLAB and Simulink provide co-simulation function and blocks which can be used by behavioral HDL model in an HDL Simulator. However, co-simulation of RTL level models with MATLAB and Simulink is not supported in this tool.

Ahmed et al. (2017) addresses incompatibility between simulation tools and model composability as future challenges in High Performance Computing (HPC) modeling and simulation. One of the solutions they suggested was to agree on a communication API for different simulation components. Also, they emphasized on creating and maintaining comprehensive model libraries of hardware and software components. In our work, we use DEVS-FMI interface as communication API between simulation components. We have added new models to the existing library of MIPS models. Our co-simulation approach and the DEVS-Suite simulator can be used to tackle the software and hardware co-design at multiple levels of abstraction using different modeling methods with co-simulation support.

4 MODELING CIRCULAR BUFFER

Buffers are an integral part of NoC design. In a System on Chip (SoC), packets sent from one IP to another are kept temporarily in a buffer when there is traffic in the network. In this section, we consider a circular buffer (also known as a ring buffer) model as an example. The buffer operates in First-In-First-Out (FIFO) manner and it has four memory locations for storing data packets. Separate read and write pointers keep track of the memory locations to be sequentially read from and written one at a time. When the last slot in buffer is read or written, the corresponding pointer points to the first slot again. The relative position of the pointers is used to determine the empty or full state of the buffer.

4.1 DEVS Modeling

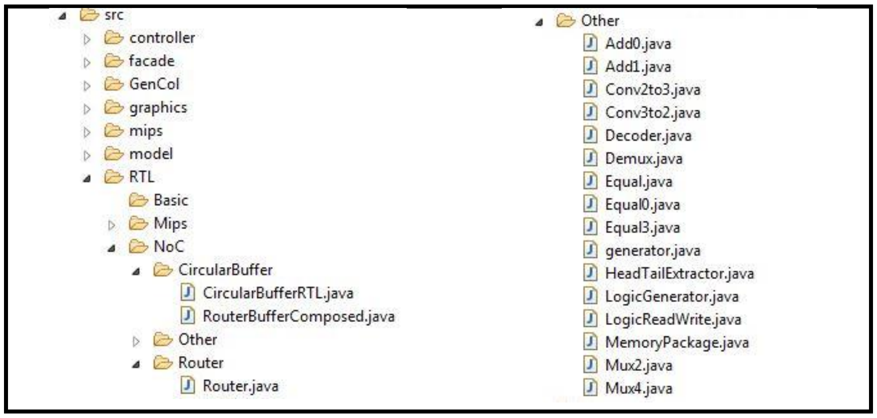
A library of previously developed RTL models (Chen and Sarjoughian 2009) are used to create a RTL circular buffer model. For modeling the Circular Buffer, other models including adder, read and write pointer multiplexers, comparator, and, or, and bit converter supporting pointers are introduced to the

DEVS-RTL model library implemented in the DEVS-Suite Simulator. Figure 1 shows a partial list of components added to the DEVS-RTL model library (see also the package named *Other* in Figure 2). In Section 4.2, this design is extended by adding a Router atomic model. As DEVS allows hierarchical design, some of the models are blackboxed into two coupled models *ReadWriteLogic* and *BufferMemory-4*. ReadWriteLogic model is created using basic gates (AND, OR, NOT) which provide the necessary logics to read and write the buffer. BufferMemory-4 model is created using registers and multiplexers and it represents the four memory slots of the circular buffer. A coarse grain model (i.e., without gate-level abstractions) for the circular buffer is also developed.

Component	**Instances**	**Functionality**
`Register With Enable)`	`memory0,1,2,3`	`Four slots for the circular buffer; each slot can hold 4 bits of data`
	`Readptr`	`stores the current pointer to the memory location to be read`
	`Writeptr`	`stores the current pointer to the memory location to be written`
	`data_out`	`stores output data while reading`
`Decoder`	`Decoder`	`Each output of decoder works as a selector for memorymux block`
`Adder`	`add0`	`increments write pointer`
	`add1`	`increments read pointer`
`Comparator`	`Equal0, Equal1`	`Compares its two inputs to see if they are equal`
`Converter`	`Conv2to3_rdptr`	`In RTL created from VHDL, a new wire is added or existing wire is grounded to support different bit sizes; models implement same functionality for DEVS-RTL; models convert two bit binary to three bit or three bit binary to two bit`
	`Conv2to3_wrtptr`	
	`Conv3to2_wrtptr`	

Figure 1: Partial list of Components added to MIPS library in DEVS-Suite Simulator.

Figure 2 shows the NoC source library added to the RTL package developed for the DEVS-Suite Simulator. *CircularBufferRTL.java* and *RouterBufferComposed.java* correspond to the coupled models shown in Figure 3 and Figure 4 respectively. Figure 2 also shows additional model libraries (Other directory) developed and used for DEVS-RTL design. This RTL circular buffer is implemented and simulated in the DEVS-Suite simulator 4.0.0 (ACIMS 2018).



```
src
  controller
  facade
  GenCol
  graphics
  mips
  model
  RTL
    Basic
    Mips
    NoC
      CircularBuffer
        CircularBufferRTL.java
        RouterBufferComposed.java
      Other
      Router
        Router.java

Other
  Add0.java
  Add1.java
  Conv2to3.java
  Conv3to2.java
  Decoder.java
  Demux.java
  Equal.java
  Equal0.java
  Equal3.java
  generator.java
  HeadTailExtractor.java
  LogicGenerator.java
  LogicReadWrite.java
  MemoryPackage.java
  Mux2.java
  Mux4.java
```

Figure 2: NoC Library added to DEVS-Suite Simulator.

4.2 Model Extension with Router

The DEVS RTL model developed in section 4.1 is extended with a router model (Figure 4) and co simulated with a software model. A MATLAB script running in MATLAB execution environment represents the software system. The hierarchical coupled model shown in Figure 4 has components at two different abstraction levels. *Router* and *HeadTailExtractor* are modeled as hardware Intellectual Property (IP). Other models are at the RTL abstraction level. The circular buffer model in Figure 3 is shown as a blackbox. The router is an atomic model which represents a microprocessor with a firmware. The function of the router model is to decide route for incoming data packets. The firmware is a software permanently programmed into a flash memory inside the microprocessor. The firmware is modeled as a MATLAB script (*RouterFirmware.m*). Details of the firmware model are discussed in Section 4.3.

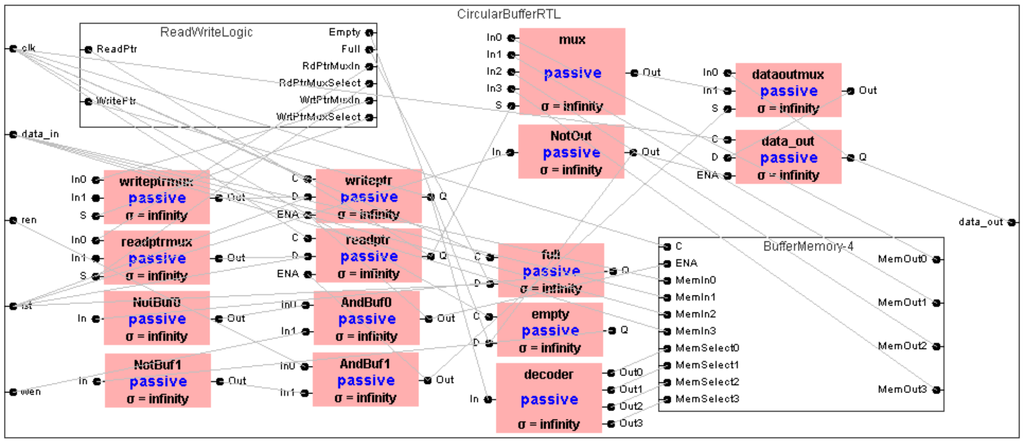


Figure 3: Hierarchical DEVS model of circular buffer.

The R, H, T, HeadTailExtractor, and Demux atomic models are added in the extended model. We designate these as supporting models for the circular buffer and the router. The R, H, and T models are status registers that store Head, Tail and destination route of a packet, respectively. HeadTailExtractor model receives a packet from circular buffer and provides head flit to the router. The Head flit contains destination address which is used for packet route calculation. The Demux model sends packets to correct output ports using the decision from the router model.

An experimental frame model consisting of three atomic models: clock, generator and logic generator is developed. The clock model provides 0 and 1 values to the *clk* input of the circular buffer denoting high and low state of clock pulse. The logic generator model provides the logic to enable the circular buffer in read or write mode. The generator creates data packets and sends them to the *data_in* input port of the circular buffer. Each packet is represented with 14 bits of data. The first 6 bits denotes head flit which has destination address. The following 4 bits are data bits and last 4 bits denote tail flit. For example, in a packet represented by 01100111110000, 011001 is the head flit, 1111 is 4 bit data, and 0000 is the tail.

4.3 Model Development in MATLAB

The router modeled in DEVS is abstracted as a microprocessor which has a firmware embedded in it. We have modeled the firmware using a MATLAB script. The script has a function which takes destination information from head flit and calculates route using adaptive x-y routing. Unlike DEVS atomic model, the script is untimed; it has no clock or time advanced function in it. It executes and returns route value when requested. In tool coupling co-simulation, an FMI adapter is required for slave tool to connect with FMI. Existing adapter is developed for MATLAB (Widl and Müller 2017) on top of FMI++ library (MATLAB-FMIPP). A new derived class is developed (*RouterFMIAdapter.m* file shown in Figure 5) from this base adapter so that firmware script can communicate and perform data exchange with FMI during co-simulation. This RouterFMIAdapter has functions to initialize firmware I/O and perform simulation steps. RouterFMIAdapter and its base adapter uses MEX (Matlab Executable) functions to communicate with C++ backend functions. Firmware model and FMI adapter are packaged into an FMU (*RouteCalculator.fmu*) using scripts developed in MATLAB and Python as part of MATLAB-FMIPP project. These scripts also create *modelDesription.xml* file for FMU. Figure 5 shows list of files when RouteCalculator.fmu is unpacked.

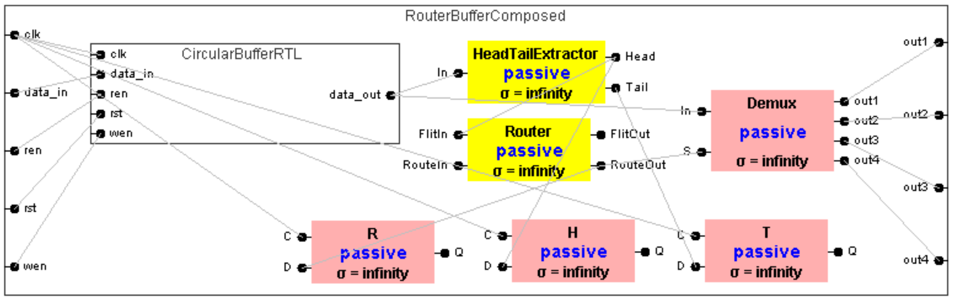


Figure 4: Circular buffer model extended with Router and supporting atomic models.

5 CO-SIMULATION OF HARDWARE AND SOFTWARE MODELS

The extended DEVS-RTL model of section 4.2 and scripts developed in MATLAB are coupled in a co-simulation environment. FMI++ (Widl et al. 2013) is used as a middleware to achieve interoperability of DEVS-Suite Simulator and MATLAB during co-simulation. We discuss the development of the interface, execution and timing aspects of the co-simulation in this section.

5.1 Development of DEVS-FMI Interface

A DEVS-FMI interface is developed for DEVS-Suite Simulator. This interface is intended for communication between simulator classes of DEVS-Suite with FMI++ functions. Figure 5 illustrates different packages created in DEVS-Suite Simulator. DEVS-FMI interface and the classes using that interface is added to simulation package of DEVS-Suite. Frontend.java class contains Java native functions required to access C++ functions of FMI++ frontend. Java Native functions along with the C++ implementation of the functions in FMI++ are compiled into a library named jni-frontend.dll (Figure 5) which is used by class Frontend. *CoSimIO.java* class implements DEVS-FMI interface and extends Frontend class to provide necessary functions for co-simulation.

In DEVS-Suite Simulator, modeling and execution layer are separated from each other as shown in Figure 5. Development of this interface benefits from this separation. Each atomic model is simulated by a dedicated atomic simulator and each coupled model is simulated by its dedicated coordinator. The simulator classes interact with FMI functions using the interface and perform necessary data exchange with modeling layer. This ensures flexibility to use different models in co-simulation without changing the DEVS-FMI interface or simulator.

5.2 Using FMI++ as Middleware for Tool-coupling

FMI++ (Widl et al. 2013) is basically the FMI 1.0 implementation wrapped with C++. There are two main utility packages in FMI++ library: Export and Import. These packages are compiled into library files fmippex.dll and fmippim.dll respectively (Figure 5). Import utility package provides functionality to import FMU and use it as a model in a simulation tool. The export utility allows existing simulator tools to export their model as FMU as well as connect to other simulator tools. In section 4.3, MATLAB models are packaged into FMU using a wrapper of the export package of FMI++. In this way, FMI supports modular development, allowing creating FMU as a model entity. When FMU is used in a standalone co-simulation or tool-coupling co-simulation, FMI basically provides the capability of distributed execution of the model embedded in FMU.

Export utility package has three key components: Frontend, Backend and Data Manager. Frontend component provides the interface for external simulation tool to connect and implement functionality as

FMI component. Backend component is the counterpart of frontend which can be used by the slave tool to connect and exchange data with the Frontend. The in-between data manager ensures proper synchronization and data handling using shared memory. Data manager uses semaphores to ensure safe access to variables. Shared variables in data manager are not read or written unless blocking functions *waitForMaster* and *waitForSlave* returns. Also, *signalToMaster* and *signalToSlave* functions are used to notify simulators after data is read or written. These three components are used to connect DEVS-Suite simulator and MATLAB during execution of co-simulation as shown in Figure 5.

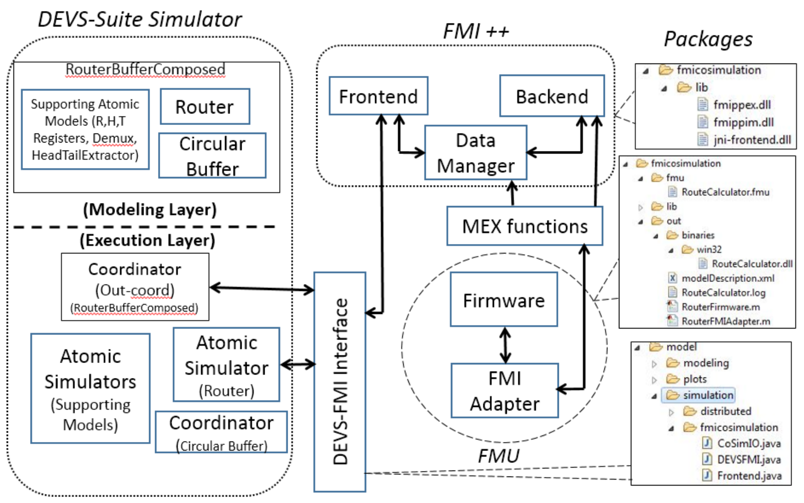


Figure 5: Illustration of co-simulation of DEVS-Suite and MATLAB using FMI++.

RTL-DEVS models in the DEVS-Suite simulator are written in Java and the software model is written in the MATLAB script language. As the DEVS-Suite simulator does not support FMI++, the DEVS FMI interface is developed. The DEVS-Suite simulator and MATLAB use C++ functions via Java Native Interface (JNI) and MEX (MATLAB executable) functions, respectively.

5.3 Execution of Co-simulation

The circular buffer can be operated in two different modes: read mode and write mode. If write mode is enabled (*wen=1, ren=0*) and the buffer is not full, packets received in *data_in* port are saved into the buffer memory. In read mode (*wen=0, ren=1*), data is read sequentially and available at the *data_out* port of circular buffer. The circular buffer is constrained to read or write only one packet at a time. When each packet is read, the Router atomic model in Figure 4 needs to provide destination route to Demux model. Demux model then sends the packet to one of the four output ports based on the route. Destination route is calculated by the firmware model during co-simulation. Therefore, in the read mode, DEVS-Suite Simulator needs to initiate co-simulation every time a packet is read.

The DEVS-Suite simulator has a hierarchical simulator architecture for its hierarchical model (Figure 5). Each atomic model is simulated by a simulator and each coupled model is simulated by a coordinator. There is a root coordinator which coordinates all the simulators and coordinators. At the beginning of the simulation, the coordinator for the outer coupled model, let's call it *out-coord*, unpacks the FMU and parses information about slave simulator tool and shared variable from modelDescription XML file. Two shared variables of type Integer are specified in the modelDescription file: Head and Route. Head is an input variable and Route is output variable. Root coordinator also has the information which atomic model takes part in data exchange during co-simulation. In the extended model we developed (Figure 4), Router atomic model takes part in data exchange in co-simulation. The atomic simulator for Router model uses DEVS-FMI interface to perform this data exchange (Figure 5).

Table 2 provides a list of key functions used by master simulator during co-simulation in order of their execution. When a packet is read, the HeadTailExtractor model sends the head to Router model. As soon as the head is received, Router model decodes destination x and y coordinate from the head. *Out-coord* uses DEVS-FMI interface to notify FMI++ Frontend to instantiate slave and initialize. The atomic simulator for Router writes the shared variable *Head* using *setInteger* function and calls the *doStep* function. At this point, MATLAB executable simulates the firmware model and updates Route variable through FMI-Adapter. Atomic simulator reads the updated Route and *out-coord* sends it Router. Router model produces the route value as output and it is sent to Demux model. Demux sends the packet to the output port.

5.4 Time Synchronization

In a co-simulation, subsystems are simulated independently in their respective simulators and data exchange takes place during simulation. The simulators are responsible for maintaining their own simulation time. When the simulators have different notion of time, synchronization of their timing is required.

Each DEVS atomic model has its own timing denoted with variable sigma and maintained by simulator using time advance function as discussed in Section 2.1. As hardware models are executed using clock steps, a separate clock model is used for fixed step execution. Models that create combinational circuit (i.e. logic gates) have a sigma of zero. Additional timing parameters, like latency and propagation delay or setup time and hold time of registers can be modeled using time advance function in DEVS. During monolithic simulation, DEVS simulator clock advances its clock according to the sigma of different models.

Slave simulator executes specified steps within a time window defined by a start and a stop time. FMI adapter of the slave is aware of a communication point when data exchange takes places during runtime. Data manager takes care of synchronization of data exchange as discussed in section 5.2. In our simple example, the firmware model developed in MATLAB has no notion of time, it is an untimed script. The script executes only once to produce route value. After running, it writes to the shared variable immediately. Therefore, the concept of start time, end time, step size and communication point is not needed for execution of the example. The execution of the script can take arbitrary time, depending on the machine that runs the simulation. Even though FMI provides means for synchronized data exchange, FMI is untimed like the MATLAB scripts. Like C++ program, execution of frontend, backend and data manager is sequential.

A sample execution for the router is shown in Figure 6. Here, trajectories show that the Router receives the head input from HeadTailExtractor on its *FlitIn* port at time t=300. When an input on *FlitIn* input port is received, its atomic simulator sends it to MATLAB via the DEVS-FMI interface. The calculated route is collected by out-coord and sent back to the router through *RouteIn* port. Upon receiving input *RouteIn* input port, the phase is changed to *SendRoute* and the route is sent to Demux via the *RouteOut* port at *t = 320.*

6 RESULT AND PERFORMANCE ANALYSIS

DEVS models are developed using Java in Eclipse (Eclipse Foundation 2001) and simulated in DEVS Suite Simulator version 3.0.0. Router adapter and firmware models were developed using MATLAB R2013a. FMI++ library functions have compile-time and run-time dependency on Boost libraries (Dawes et al. 1998). Boost version 1.58.0 is used. The co-simulation is performed on a Windows 7 machine with Intel Core2Duo 2.93 GHz CPU and 8 GB of RAM.

Using the above environment, the actual time recorded for executing the circular buffer using the DEVS-Suite Simulator and FMU is tabulated in Table 3. Hierarchy 1 refers to an RTL model is a coupled model that has only atomic models. In Hierarchy 2, selected atomic models of are represented as hierarchical models. Simulation time is recorded for monolithic simulation of circular buffer model and also for co-simulation with MATLAB. As expected, hierarchical models require more time to execute due to additional I/O couplings and message passing. Also, we observe that co-simulation takes more time than monolithic simulation. The model developed for co-simulation has a router and other supporting models added to the circular buffer model. Execution time of these additional models add to the simulation time.

Moreover, in co-simulation, time to instantiate slave, run steps in slave simulator and data exchange in FMI requires more computational resources and thus require additional time to execute.

Table 2: List of key functions used in FMI for co-simulation in order of execution.

FMI Function	Purpose
instantiateSlave	Master invokes slave simulator to initiate co-simulation
initializeSlave	Initialize slave with timing information (start time, stop time)
setInteger	Set the value of shared input Integer variable
doStep	When master calls doStep function, slave performs simulation steps specified by start time, stop time and number of steps.
getInteger	Get updated value of shared output variable after simulation

Table 3: Simulation times for monolithic simulation and co-simulation of the circular buffer model.

Simulation Type	Model	Simulation Time (seconds)			
		100 Steps	1,000 Steps	5,000 Steps	10,000 Steps
Monolithic (DEVS-Suite)	Hierarchy1	0.54	3.35	28.74	116.35
	Hierarchy2	0.49	3.35	29.04	118.94
Co-simulation (DEVS-Suite, FMU)	Hierarchy1	0.87	5.43	46.51	160.55
	Hierarchy2	0.86	5.60	47.63	165.59

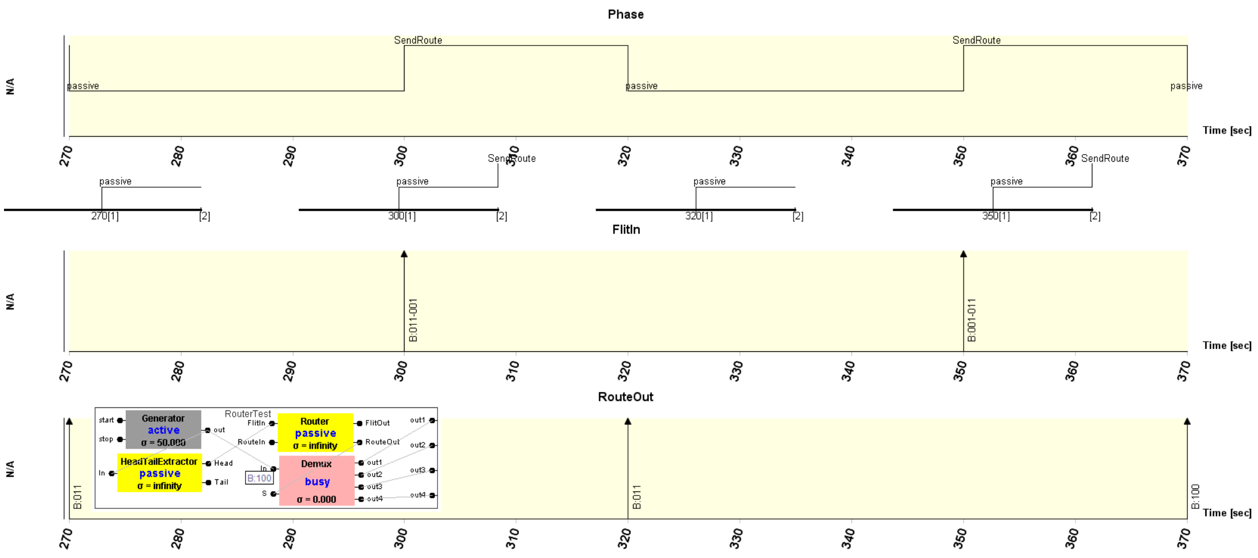


Figure 6: Superdense TimeView trajectories showing I/O communication of Router.

7 OBSERVATION

In this paper, we have demonstrated a co-simulation of hardware RTL DEVS model with a software system using FMI. In DEVS Suite Simulator, we modeled circular buffer at RTL level and a router abstracted as a hardware IP having a firmware counterpart. The firmware is abstracted as MATLAB script and our software system is represented by MATLAB script running in MATLAB execution engine. We coupled DEVS-Suite Simulator with MATLAB and thus support co-simulation of RTL and software system. Existing HDL simulators support co-simulation with software system at behavioral abstraction level.

MATLAB is primarily developed for matrix algebra and vector calculation using scripts. DEVS, however, is a time and event-driven modeling method. The firmware model developed in MATLAB is untimed. If we want to co-simulate with the software system that has a different mechanism of keeping time, it adds complexity to the design and execution. Coupling DEVS-Suite Simulator with MATLAB can be viewed as a first exemplar step to show co-simulation of RTL and Software System. This work can be extended to support coupling with tools that provide formal modeling and timing specifications.

As mentioned in Section 4.1, the RTL model in DEVS is developed using the MIPS library and some newly developed RTL models. For large-scale designs, automatic support for generation of RTL for FPGA from RTL DEVS models is key and remains as future work.

8 CONCLUSION

This paper presents DEVS modeling at System and RTL abstraction levels and co-simulation of RTL model with the software system. This is useful for co-simulating NoC system that include software. The developed proposed approach introduces FMI to support hybrid DEVS Suite Simulator and FMUs. The prototype DEVS-FMI interface developed for DEVS-Suite demonstrates lightweight interoperability with other tools. We observe that the model in MATLAB is a simple untimed script and coupling it with the DEVS-Suite presents hybrid FMI-based models and co-simulation. Ongoing research includes development of automation support for DEVS RTL from system level DEVS abstractions as well as enabling mapping from DEVS RTL to VHDL RTL and thus FPGA. Such model transformation supported with FMI is likely to help avoid some barriers associated with simulation-based design for NoC, SoC, and Internet of Things.

ACKNOWLEDGMENTS

We thank the anonymous referees for their valuable comments on an earlier version of this manuscript.

REFERENCES

ACIMS. 2018. *DEVS-Suite Simulator 4.0.0.* https://acims.asu.edu/software/devs-suite/.

Ahmed, K., J. Liu, H. Badawy, and S. Eidenbenz. 2017. “A Brief History of HPC Simulation and Future challenges”. In *Proceedings of the 2017 Winter Simulation Simulation Conference,* edited by W. K. V. Chan et al., 419–430. Piscataway, New Jersey: IEEE

Blochwitz, T., M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, et al. 2011. “The Functional Mockup Interface for Tool Independent Exchange of Simulation Models”. In *Proceedings of the 8th International Modelica Conference,* March 20th-22nd, Technical University, Dresden, Germany, 105–114.

Camus, B., V. Galtier, and M. Caujolle. 2016. “Hybrid Co-simulation of FMUs using DEV&DESS in MECSYCO”. *Symposium on Theory of Modeling & Simulation (TMS-DEVS)*, April 3rd-6th, Pasadena, CA, 1–8.

Camus, B., T. Paris, J. Vaubourg, Y. Presse, C. Bourjot, L. Ciarletta, and V. Chevrier. 2018. “Co-simulation of Cyber-Physical Systems using a DEVS Wrapping Strategy in the MECSYCO Middleware” *Simulation.*

Chen, Y., and H. S. Sarjoughian. 2009. “A Component-Based Simulator for MIPS32 processors” *Simulation* 86(5-6):1–18.

Dally, W. J., and B. Towles. 2004. *Principles & Practices of Interconnection Networks.* Morgan Kaufmann.

Dawes, B., D. Abrahams, and R. Rivera. 1998. Boost C++ Libraries. http://www.boost.org/.

Eclipse Foundation. 2001. Eclipse. https://www.eclipse.org/.

Fard, M. D., and H. S. Sarjoughian. 2015. “Visual and Persistence Behavior Modeling for DEVS in CoSMoS”. In *Proceedings of the Symposium on Theory of Modeling and Simulation: DEVS Integrative M&S Symposium*, April 12th-15th, Alexandria, United States.

Gholami, S., and H. S. Sarjoughian. 2017. "Modeling and Verification of Network-on-Chip using Constrained-DEVS". *Spring Simulation Conference.* Virginia Beach, VA, USA. 1–12.

Ghosh, S. 2001. "P2EDAS: Asynchronous, Distributed Event Driven Simulation Algorithm with Inconsistent Event Preemption for Accurate Execution of VHDL Descriptions on Parallel Processors". *IEEE Transactions on Computers* 50(1):28–50.

Intel FPGA. 2018. Quartus® Prime. https://www.altera.com/products/design-software/overview.html.

Maillet-Contoz, L., and J. Strassen. 2005. "TLM Modeling Techniques". In *Transaction Level Modeling with SystemC*, 57–94. Springer.

Mano, M., C. Kime, and T. Martin. 2015. *Logic & Computer Design Fundamentals.* Pearson.

MathWorks. 2017. HDL Verifier. https://www.mathworks.com/products/hdl-verifier.html.

Ni, N., M. Pirvu, and L. Bhuyan. 1998. "Circular Buffered Switch Design with Wormhole Routing and Virtual Channels." *International Conference on Computer Design: VLSI in Computers and Processors.* IEEE, October 5th-7th, Austin, Texas, USA, 466–473.

Park, J., B. Lee, K. Lim, J. Kim, S. Kim, and K. Baek. 2008. "Co-simulation of SystemC TLM with RTL HDL for Surveillance Camera System Verification". In *15th IEEE International Conference on Electronics, Circuits and Systems*, August 31st-September 3rd, St. Julien's, Malta, 474–477.

Sarjoughian, H. S., A. Alshareef, and Y. Lei. 2015. "Behavioral DEVS Metamodeling". In *Proceedings of the 2015 Winter Simulation Conference*, edited by L. Yilmaz et al., 2788–2799. Piscataway, New Jersey: IEEE.

Seok, M. G., T. G. Kim, C. B. Choi, and D. Park. 2017. "An HLA-Based Distributed Cosimulation Framework in Mixed-Signal System-on-Chip Design". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25(2):760–764.

Shiva, S. G. 1979. "Computer Hardware Description Languages—A Tutorial". *Proceedings of the IEEE* 67(12):1605–1615.

VHDL. 1988. "Language Reference Manual." *IEEE Std*. 1076–1987.

SystemC. 1999. Accellera Systems Initiative. http://www.accellera.org/.

Vivado Design Suite. 2018. Xilinx Inc. https://www.xilinx.com/products/design-tools/vivado.html.

Widl, E., and W. Müller. 2017. "Generic FMI-compliant Simulation Tool Coupling". *The 12th International Modelica Conference*, May 15th-17th, Prague, Czech Republic, 321–327.

Widl, E., W. Müller, A. Elsheikh, M. Hörtenhuber, and P. Palensky. 2013. "The FMI++ library: A High-Level Utility Package for FMI for Model Exchange". *Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES).* Berkeley, CA. 1–6.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* Second Edition. Academic press.

AUTHOR BIOGRAPHIES

MASUDUL H. QURAISHI is a Computer Engineering PhD student at ASU. His area of interests are Modeling and Simulation, Cyber-Physical Systems, Digital Design, and FPGA. He can be contacted at mquraish@asu.edu .

HESSAM S. SARJOUGHIAN is Associate Professor of Computer Science & Computer Engineering at Arizona State University and Co-Director of the Arizona Center for Integrative Modeling and Simulation. His research focuses on model theory, polymorphic model composability, distributed co-design modeling, visual simulation modeling, real-time modeling, and service-oriented simulation. He can be contacted at sarjoughian@asu.edu.

SOROOSH GHOLAMI received his PhD degree in Computer Science at Arizona State University in December 2017. Currently he is working as a Software Engineer at Paypal Holdings Inc. He can be contacted at soroosh.gholami@gmail.com.