CO-SIMULATION OF C-BASED SOC SIMULATORS AND MATLAB SIMULINK

Frank Poppen  Kim Grüttner

OFFIS – Institute for Information Technology  OFFIS – Institute for Information Technology
Escherweg 2, 26121 Oldenburg, Germany  Escherweg 2, 26121 Oldenburg, Germany
frank.poppen@offis.de  kim.gruettner@offis.de

ABSTRACT

Simulation of systems under development is a widely used methodology for early design evaluation and performance analysis. Many engineers trust on MATLAB & Simulink as a simulation environment, especially since it offers many domain specific block sets for fast, easy and efficient use. With its ability to generate life source code from the simulation model it becomes a powerful development tool. After code generation it is common practice to proceed to the real world and maybe coupling HW executing the generated SW with simulation (HW in the loop simulation), which can be considered to be a big jump into the cold water. The contribution of this paper is a concept and proof-of-concept implementation of a co-simulation interface between a C-based System on Chip (SoC) model and MATLAB & Simulink. The proposed approach enables the coupling of application domain specific high level simulation with a bit and cycle accurate virtual execution platform of a specific embedded HW/SW platform without interfacing troubles. Our concept was implemented for and applied to the development of an embedded medical device – a Wearable Artificial Kidney Device (WAKD).

Keywords: loosely coupled, co-simulation, C-based, SoC simulation, MATLAB & Simulink

1 INTRODUCTION

Development of personal computer SW is a task that can be handled completely on a PC with compiler and debugger. A binary can instantly be executed and evaluated in the environment it is designed for – the PC’s operating system. For embedded SW on embedded systems there is a different story to tell. Embedded SW is not developed on the embedded system itself. A PC plus cross-compiler is required. To execute the binary it needs to be uploaded on the embedded system first and this is where evaluation and verification can become a real headache. “Embedded” means that these systems are part of a bigger maybe very complex environment that they should monitor and control. During verification and validation it can be impossible to control the environment in such a way that repeatable regression tests are possible.

Simulation methodologies as sketched in Figure 1 for embedded system and SW design are common practice today. Initially, the system is implemented as a model together with an abstraction of the environment. Ideally code is generated from this model automatically or created manually, and then possibly executed on the embedded HW in a HW in the loop simulation (HIL). Ledin (2001) documents the state of the art in this respect. HIL already excludes the hassles one would run into to create reproducible test cases in a real physical environment. Though, using a real physical embedded HW platform makes verification, debugging and validation a finicky undertaking. The jump from a clean simulation environment to HW in the loop simulation is still quite far. Debugging at this level already includes troubles with electric signals, shorts, open connections, the use of an oscilloscope, etc. So it can be desirable to have something in between.
The implementation presented in this paper enables this intermediate step called virtual platform in the loop simulation (Figure 2). Simulation or, in other words, virtualization of HW is a widely used methodology. Commercial and free environments exist to simulate even complete PCs that can execute complex operating systems. Nanda and Chiueh (2005) give an overview on virtualization with a focus on personal computers and Linux and Windows as operating systems. As stated before the focus in this work is on embedded microcontrollers which commonly are virtualized by Instruction Set Simulators (ISS) as part of configurable System on Chip (SoC) simulations. The challenge is now to synchronize two independent simulations so that the simulated time does not diverge and data can be exchanged at the right instant of time.

In this paper we present a co-simulation infrastructure that can be used to couple MATLAB & Simulink with C-based virtual platforms, thus establishing a virtual platform in the loop environment. As an early proof of concept the idea was refined by applying it to a combustion engine control system (Fakih et al., 2011). In Pielawa et al. (2011) we made good experience with the methodology in the medical domain. More related work is presented in Section 2.0. In the following a quick introduction to the general concept of co-simulation and the proposed co-simulation link is provided. In Section 3.0 we define the synchronization scheme of a loosely coupled co-simulation approach. Section 4.0 shows the general implementation of three required C-functions to interface with MATLAB & Simulink. In Section 5.0 the integration of these three C-functions into the Open Virtual Platform (OVP, www.ovpworld.org) environment is presented. A use-case demonstrates the applicability of our proposed co-simulation approach in Section 6.0. The conclusions are to be found in Section 7.0.

2 RELATED WORK

Other authors handled the coupling of SoC simulation with MATLAB & Simulink before. In this section we give a swift overview. Boland et al. (2005), Hassairi et al. (2009) and Mühleis et al. (2011) present co-simulation interfaces between SystemC (Master) and MATLAB & Simulink (Slave) very similar to this paper. Their intended purpose is functional verification of designs at different abstraction levels. Boland et al. (2005) applied their concept to reuse the Simulink golden models in a parallel simulation for their Digital Signal Processing (DSP) designs and flows which is their main focus. While Hassairi et al. (2009) concentrated on a use case with a JPEG encoder scenario.
Bouchhima et al. (2006) introduce a tool called CODIS for the automatic generation of three types of co-simulation interfaces: full synchronization mode, predictable events mode, and events-driven mode. Instead of fixed-step simulation they handle synchronization with a variable step solver. Unlike our fully concurrent approach, the execution of the simulation environments is alternating in a mutual exclusive execution order. The same counts for Hylla et al. (2008). Their co-simulation control is handled by Simulink as master. Wrappers implemented in MATLAB s-functions communicate with the SystemC slave simulation partner. The work of Mendoza et al. (2011) has a similar approach, where the time synchronization scheme is controlled by Simulink and allows for a true event-based simulation inside the SystemC sub-system. Andrea Bartolini et al. (2010) proposed solution is based on the commercial instruction set simulator Simics and MATLAB & Simulink. The co-simulation of Tomasena et al. (2009) interfaces SystemC and MATLAB without specifically handling interaction with the Simulink environment. They enable two different design teams to work each in their own language (SystemC and MATLAB M-file) on the same application. For this purpose they proposed an abstraction layer for the MATLAB API which is a C++ library called MatlabEngine++. While most of this related work made use of SystemC, this paper is, to the best of our knowledge, the first co-simulation of MATLAB & Simulink and the OVP SoC simulator, OVPsim.

Harnessing the state-of-the-art from the above publications, we implemented a scenario of HW/SW simulation, where the SW model requires a high resolution in time in the range of milliseconds while the environment model of the patient physiological simulation progresses in the range of minutes to hours.

3 CONCEPT OF CO-SIMULATION

The concept presented in this paper is a loosely coupled co-simulation approach of C-based System on Chip (SoC) simulation frameworks integrating an Instruction Set Simulator (ISS) and the MATLAB Workspace from Mathworks. “Loosely” refers to the idea that the two execution models run independently in parallel for a defined time period after which they synchronize (compare with the OSCI TLM-2.0 standard (www.systemc.org) for SoC shared bus communication modeling). To avoid any misunderstandings in the remainder of this work a clear separation between the Wall Clock Time (WCT, ordinate of Figure 3) which is the simulated time and the time it takes to execute the model on a host machine named Model Execution Time (MET, abscissa of Figure 3) is maintained.

![Figure 3 Loosely coupled co-simulation of models with imbalanced and changing execution times requires synchronization](image)

If WCT is smaller than MET the simulation is running slower than the real world, while if WCT is larger than MET the simulation is running faster than real time. The co-simulation concept in this
paper is required to ensure that the \( WCT_M \) (WCT of MATLAB Workspace) proceeds at the same speed as \( WCT_I \) (WCT of the ISS) guaranteeing that \( WCT_M = WCT_I \) whenever the two environments are synchronized. It is usually the case that \( MET_M \) and \( MET_I \) are not equal (\( MET_M \neq MET_I \)) and can even constantly change depending on the complexity of the model. After each defined synchronization period (\( WCT_{sync} \)) simulation has to check and when required wait for the second model to catch up (Figure 3).

The concept of a loosely coupled co-simulation is introducing imprecisions to the simulation as each simulation environment is unaware of potential influences form the other between synchronizations. Events will be delayed or even stay unrecognized if the same event is changing more than once between synchronization. The created error is potentially reduced to zero if an appropriate \( WCT_{sync} \) is chosen. This is possible when the influence of the embedded system on the environment is discretized by sensors and actuators. If \( WCT_{sync} \) is smaller than the simulated sensor sampling frequency the introduced simulation error is masked by the system’s sampling error. The same counts for the actuators if their control parameters depend on sampled sensor data, and thereby are discretized as well.

4 USING MATLAB’S ENGINE.H API

The implementation of our concept bases on MATLAB’s `engine.h` interface. The C-based SoC simulation becomes the master by using this interface and controlling the MATLAB Workspace and Simulink (starting MATLAB, opening Simulink model, starting/ending/continuing paused Simulink simulations, creating/changing Workspace variables).

Our implementation requires the Simulink model to pause simulation after each \( WCT_{sync} \) to synchronize with the SoC environment. This has been achieved by using fixed-step simulation mode and including an assertion in the Simulink model. Function `fcn` in Figure 4 is a simple \( y = \text{mod}(u, 1) \) to generate a synchronization pulse for the assertion: `set_param(bdroot, 'SimulationCommand','pause')`.

![Figure 3 Simulink pauses for each synchronization period](image)

The example above assumes that Simulink makes fixed steps of 1 second WCT and synchronizes each second (\( \text{mod}(1) \) generates pulses at every step). Different configurations are possible. One simulation step e.g. could be interpreted as 1 ms, and a second-by-second synchronization would require modulo 1000 so that Simulink pauses every 1000th step. The required resolution of this synchronization scheme depends on the modeled application. While synchronization in the range of ms would be appropriate for a combustion engine control model, a minute-by-minute synchronization is efficient for slow changing models of e.g. the human physiology in a blood cleansing (dialysis) scenario.

In the Simulink model an embedded MATLAB function has been used to read values (actuators) from the Workspace `fout=evalin('base','variableName');` and the To Workspace block from the Simulink library to write values (Sensors) to the Workspace. All that now remains is to read/write the workspace from the software executed on the ISS and continue Simulink simulation after each time that it has paused. For this we created three functions.

4.1 C-Function startMATLAB

This function is called once at startup of the SoC model. It uses MATLAB’s API (1) to start the MATLAB environment (2), changes into the directory of the Simulink model (3) and opens it (4). Hereafter the Simulink model will be in the stopped state at \( WCT_M = 0 \).
4.2 C-Function syncMATLAB

The implementation of the SoC model has to track its own time WCT\textsubscript{1} to call syncMATLAB after each WCT\textsubscript{sync} period (details in 5.2). When called, the function checks the status of Simulink simulation (1) by a) having the MATLAB environment compare the parameter `SimulationStatus` with the string `stopped` using the MATLAB function `strcmp` and store the boolean result in the MATLAB workspace. This Boolean is received using b) the engine function `engGetVariable()` and extract from the received mxArray the boolean value required. The status `paused` can be retrieved in the same way. If the status is stopped, the Simulink model was just opened using the above mentioned function `startMATLAB`. If it is paused, previous synchronization events have already occurred. From a synchronization point of view these two states do not make any difference. But it is necessary to know for Simulink to continue, which is either `start` (when stopped) or `continue` (when paused).

For reading sensor variables from the workspace we proceed as described in (1) using `engGetVariable` and `mxGetScalar`. For writing actuator variables to the workspace we use (3).

If the simulation is neither stopped nor paused, it is assumed, that Simulink is still busy. The function waits and tries again later (4) in a polling loop.

4.3 C-Function closeMATLAB

The only activity this function implements when the C-based SoC model terminates is (1):

5 IMPLEMENTATION WITH C-BASED SOC MODEL

The concept presented in Section 2 has been implemented using MATLAB’s API as described in Section 4. The co-simulation with C-based SoC models has been implemented for two different SoC simulation environments: the SystemC based SoCLib\textsuperscript{1} (Fakih et al., 2011) and the C based OVP from Imperas (Pielawa et al., 2011). In the following we take a close look at the implementation for OVPsim.

5.1 OVP Model

This section provides a rudimentary introduction on how to create platform models with OVP. OVP provides a library to model SoC using regular C-programming language. After compilation of the model with e.g. the GNU compiler gcc the resulting executable resembles the SoC simulation. This executable program is depicted as dark blue box in Figure 5. To simplify matters we assume that the platform already includes any microcontroller model as e.g. an ARM ISS, as well as a 32 bit system.

\textsuperscript{1} www.soclib.fr
bus. For all the details please refer to the OVP documentation Imperas (2010). In the following we take a detailed look at how the OffisSimLink peripheral model is instantiated and used.

In (1) the function icmNewPSE returns a pointer to the instantiated bus peripheral model with the name “OFFIS_simlink”. This name is for debugging purposes only. The second parameter specifies the path and name to the peripheral’s PSE model (light blue box “OffisSimLink (x86 ISS) in OVP Model of Figure 5, further details next Chapter 5.2) while the fourth parameter points out path and name to the host native code in the Semihost.dll. The latter implements the C-functions as introduced in 4.0, while the first implements only empty dummies. OVPsim offers the concept of “intercepting” functions during simulation so that these are not executed within the OVP simulation environment (dark blue box of Figure 5), but outside, directly on the host platform. Here, the Semihost.dll is able to access MATLAB.

The third parameter Attr is a list of configuration parameters (2) for the peripheral. (2) shows how the function icmAddUns64Attr is used to add the attribute “simSyncInterval” with the value of simSyncInterval to the list Attr. Other configurable attributes of OffisSimLink are strlenSimulinkModelPath, simulinkModelPath, strlenSimulinkModelName, simulinkModelName. These attributes contain the necessary information to be forwarded to the Semihost.dll to operate MATLAB.

Finally, the peripheral is connected to the bus-model (3) and interrupt line (4). In the following we will skip the implementation of the interrupt line to keep things focused. An Interrupt is to be understood as a special case of data communication from Simulink to OVPsim and realized similar to the following discussion. By this means the Simulink environment is able to trigger interrupt signals to the ISS.

In our example OffisSimlink is mapped to the address space from 0x80010000 to 0x80010054 (84 bytes = 10 actuators plus 10 sensors at 4 bytes (32 bit) each, and one irqVector of 4 bytes). Whenever the ISS or “Hello.c” program respectively accesses an address in this range, data will be read/written from/to the OffisSimLink and thereby consequently from the Simulink model.

Figure 5 OVP-Simulink co-simulation architecture

In our example OffisSimlink is mapped to the address space from 0x80010000 to 0x80010054 (84 bytes = 10 actuators plus 10 sensors at 4 bytes (32 bit) each, and one irqVector of 4 bytes). Whenever the ISS or “Hello.c” program respectively accesses an address in this range, data will be read/written from/to the OffisSimLink and thereby consequently from the Simulink model.
5.2 OFFISSIMLINK (X86 ISS)

The OVP documentation Imperas (2010) gives a good description on how simulation bus peripheral models are implemented. In this chapter only a quick introduction to OVP’s concept is given. Bus peripherals in OVP are not created and simulated as such. Instead, each peripheral is represented by its own x86 ISS. A compiled program (the PSE model) executed on this x86 ISS is mimicking the behavior of the desired peripheral. In the following the C-code (offisSimLink.c) that we compile as PSE model and execute on this x86 ISS to mimic the OffisSimLink peripheral is introduced.

(1) defines the bus slave port of our peripheral so that it matches to the previously described connection in name and size. This goes along with the definition of the interrupt net in (2). In (3) one of the three empty dummy functions is declared that should never be executed inside the SoC simulation. As already mentioned, whenever the x86 ISS is about to call one of these functions, the execution will be intercepted and passed through to the Semihost.dll running natively on the host machine.

Internal registers of OffisSimlink are defined by (4). The example shows with parameter one the creation of the irqVectAddr register. Second parameter has a descriptive debugging functionality. Third parameter names the port through which the register can be accessed. Fourth parameter defines the offset of the defined register from the base address, while the fifth specifies it size in bytes (bytes 0 to 3 starting from 0x80010000). For our implementation the first actuator register is therefore at bytes 4-7 and the first sensor 8-11 etc. The next parameters reference the read callback function and the write callback function. Whenever the simulation detects an according access, these functions will be executed. Reading from irqVectAddr does not do any meaningful, but writing to this address clears the IRQ signal (5).

The code introduced until now is executed once during initialization of the peripheral. For runtime behavior an infinite task running on the x86 ISS needs to be created (6) and (7). The infinite loop of (7) contains the periodic call of the syncMATLAB function introduced in 4.0, as well as a bhmWaitDelay to execute it only every period of WCTsync as introduced in Section 3.0. This period needs to be in line with the Simulink pause-intervals (Section 4.0). Additionally it is checked, if the value of irqVectAddr does not equal zero. The Simulink model will set a value to indicate an interrupt. In this event the value 1 is written to the interrupt net. As already mentioned, the interrupt can be reset by writing to the irqVectAddr register (5).

6 EVALUATION OF CONCEPT BY MEDICAL USE CASE

The OffisSimLink co-simulation interface was developed within the EU funded Nephron+ project (see Section Acknowledgements) and applied to the development process of a Wearable Artificial
Kidney Device (WAKD). The WAKD is a highly complex multi-nature ICT (Information and Communication Technology) system that includes fluidics, chemistry, electrics and electronics in a medical application to control the human physiology (Pielawa et al., 2011). Being a central part of this, the implementation of the control software requires an elaborate strategy for parameterization, verification and validation. Part of this strategy relies on intensive system simulations. The system and the patient are implemented as a model of computation. If the abstraction of simulation is such that it is still accurate enough to depict the relevant aspects of the system, it has many advantages, which are (1) reproducible test scenarios, (2) fast execution, (3) shortening implementation time and as a combined effect of the first (4) safety.

6.1 Results achieved

At the highest level of abstraction with the lowest level of detail the WAKD was divided into 1) WAKD Control, 2) WAKD Fluidics including sensors & actuators and 3) Patient Physiology. All three parts of the NEPHRON+ system are initially modeled in Simulink. Hence, simulation is executed in one single simulation environment.

Following a V-model development process, WAKD control is refined into hardware and software functionality. The SW itself is separated into the operating system FreeRTOS\(^2\) and the WAKD functionality specific SW tasks. Mathworks Real Time Workshop\(^3\) technology was applied to automatically generate the SW code basis directly from the Simulink model. At this step of the development the real-time behavior of the SW in combination with the chosen micro-controller (ARM7TDMI) becomes of special interest. The Simulink environment is not capable to simulate the execution of embedded software at the necessary cycle accurate level of accuracy. This is the domain of Instruction Set Simulators (ISS).

Figure 6 shows how the OffisSimLink co-simulation concept was applied to the Nephron+ use case. The blocks of “FreeRTOS” and “Nephron+ SW Tasks” in the same figure no longer stand for a model, but symbolize the (final) version of the embedded SW to be run later as is on the real micro-controller.

![Co-simulation of WAKD Control with Fluidics and Patient Physiology model](image)

Instruction accurate simulators, such as OVPsim, simulate only the instruction behavior of the processor, and typically have no timing information. External memories are simulated, but are assumed to be ideal, and no latency information is provided. Caches are by default not modeled. As in the case of OVP the processor pipeline is not emulated. Due to the efficient nature of the binary translation simulation engine in translating target processor instructions to host instructions, the OVP

\(^2\) www.freertos.org
\(^3\) www.mathworks.com/products/rtw
model enables the instruction by instruction accurate simulation of SW execution on the targeted ARM processor at about four times the speed of real time for our project. Hence, one hour of therapy is simulated in fifteen minutes. This opens up the opportunity for extensive evaluation runs of the implemented embedded SW prior to its deployment on the slower and harder to debug embedded HW.

6.2 Benchmark for varying WCTsync

As mentioned in 3.0 the concept of a loosely coupled simulation introduces an error that depends on the relation of the simulators’ synchronization period WCTsync to the sampling frequency f of simulated sensors and actuators. WCTsync must be smaller than f to prevent significant errors in simulation results. The technology implemented in this work is part of the Nephron+ wearable artificial kidney project, where physiological sensors for urea, potassium and sodium measurements are sampled every minute. According to the data presented in Figure 7 the synchronization overhead is less than 5% with a simulation performance of more than four times real-time. A second more detailed model of the WAKD requires the assessment of physical system sensor data (pressure, conductivity, temperature, pump-speeds, voltages, …) which are sampled at a period of one second. In correlation to the data of Figure 7 we achieved a performance of close to three times real-time. Obviously we do not recommend reducing WCTsync to less than one second since performance drops exponentially. We draw the conclusion that our implemented loosely-coupled simulation is ideal for the evaluation of embedded software within a slowly responsive environment.

![Figure 7](image)

6.3 Work in progress

The virtual prototype in the loop simulation as shown in Figure 6 is the intermediate step towards a hardware in the loop simulation (compare with Figure 2). This transition again can be done in a stepwise approach as shown in the schematic to the left of Figure 8. Once physical actuators (pumps, fluidic switches) and sensors (pressure, flow, …) are available it is possible to utilize the Mathworks HIL interface to connect the co-simulation to physical equipment. This scenario is a combination of a virtual prototype in the loop together with real HW in the loop simulation. With this configuration it is possible to extensively evaluate this HW in its interplay with the controlling SW.
Figure 8 Left: Combined Virtual Prototype and HW in the Loop Simulation. Right: State of the Art HW in the Loop Simulation

The right side of Figure 8 depicts the state of the art Mathworks HIL simulation, where extensive tests can be executed without endangering the wellbeing of lab animals or human probands.

7 CONCLUSION

We demonstrated that we were able to create a co-simulation interface that perfectly serves the needs for the development flow of a wearable artificial kidney specifically, but is applicable to the development of medical and other devices in general as well. In this respect we succeeded in bringing together HW/SW simulation which requires a high resolution in time in the range of ms, with patient physiological simulation at a resolution of seconds or even minutes to hours. This paper should enable the reader to repeat and adapt own loosely coupled co-simulation environments for virtual prototypes in the loop scenarios in the medical domain, but also for any other domain with diverging simulation time requirements as e.g. environment control (airconditioning) and certain applications of factory control (control of chemical reactors, waste water clearance). For free use and an even deeper understanding we provide the OVP OFFISSimLink peripheral for download at www.systemsynthesis.org with the publication of this paper.

ACKNOWLEDGMENTS

The author of this paper acknowledges the support by the 7th Framework Program ICT-4-5.1 for Personal Health Systems of the European Community (grant agreement: 248261) for the Nephron+ project for which the OfisSimLink peripheral was implemented. In the same context the author has to recognize the willingness of Imperas to provide the OVPsim environment under a research license.

REFERENCES


**AUTHOR BIOGRAPHIES**

**FRANK POPPEN** graduated with a degree in computer science at University of Oldenburg, Germany in 1999. Since then he is working for the OFFIS Institute for Information Technology in the Transportation Division but is also working within projects of the Health and Energy Divisions of OFFIS. As a research engineer with more than ten years of experience in EU, national and privately funded projects, his expertise is in HW design methodologies (electron design automation (EDA) tools from Synopsis, Cadence, Mentor, Xilinx, Altera, and others) with a special focus on low power SoC design. In 2007 he became the Technical Chairman of Synopsys User Group Europe (SNUG Europe).

**KIM GRÜTTNER** is manager of the “Hardware/Software Design Methodology” group at OFFIS – Institute for Information Technology in Oldenburg. Since 2005 he holds a Diploma degree in Computer Science from the University of Oldenburg, Germany, joined OFFIS in 2005 as researcher and works since 10/2008 as group manager. He is currently working towards his PhD on “Application Mapping and Communication Synthesis for Object-Oriented Platform-Based Design”. He is IEEE member and in the program committee of ESLsyn and SORT conferences. His research topics are Electronic System-Level design and synthesis for System on Chips, including system-level design methodologies and languages for HW and SW systems with a focus on HW/SW communication and interface synthesis.