M&S-BASED DESIGN OF EMBEDDED CONTROLLERS ON NETWORK PROCESSORS

Rodrigo Castro[1], Iván Ramello[2], Matías Bonaventura[1], Gabriel A. Wainer[3]

[1] Departamento de Computación
Universidad de Buenos Aires.
FCEyN. Ciudad Universitaria, Pabellón I.
(1428) Buenos Aires. Argentina.
{rcastro,abonaven}@dc.uba.ar

[2] Departamento de Computación
Universidad Nacional de Rosario
FCEIA. Pellegrini 250
(2000) Rosario, Argentina.
iramello@fceia.unr.edu.ar

[3] Department of Systems and
Computer Engineering.
Carleton University. 1125 Colonel
By Dr. Ottawa, ON. Canada.
gwainer@sce.carleton.ca

Abstract
We introduce advanced techniques to develop embedded real-time controllers for networking applications using a Modeling and Simulation (M&S) based methodology. Our solution relies on the DEVS formalism and the Embedded CD++ (ECD++) real-time simulator. We show how DEVS-based prototypes can be embedded in the target hardware, consisting of an Intel IXP2400 Network Processor. We developed interface libraries allowing DEVS models to interact with specialized microcontrollers for high performance packet handling. We also introduce a portable Virtual Lab for developing prototypes and deploying them for quick validation on a real-world network. Our approach provides model continuity, eliminating the need for adapting logic or structure of the controllers when evolving from standalone simulations to execution in the target platform.

1. INTRODUCTION

Although the software engineering community has attempted to define and use formal methods for developing embedded real-time systems, attempts have not been successful. Most existing formal methods are still difficult to scale up. Instead, modeling and simulation (M&S) techniques have shown to be a cost-effective approach to solve this problem. Nevertheless, when a project evolves toward the final target platform, most initial models and simulations are abandoned [1]. To deal with these issues, we introduced an approach based on the DEVS formalism [2]. The method combines the advantages of a practical approach with the strictness of a formal method: models initially used for specification, simulation, and verification are preserved for validation and implementation [3].

Our goal is to provide a procedure completely based on DEVS models, to be embedded in the target hardware, run in real-time, and can be connected to external hardware transparently. The strategy provides model continuity by encapsulating the interaction with external hardware in special atomic models (Mappers) that are capable of interacting with control interfaces (Drivers) for each device. To achieve our goals, we used the DEVS-based ECD++ (Embedded CD++) M&S tool [4], and embedded it on an Intel IXP2400 Network Processor (NP). Then, we equipped the software with interface libraries for communication between the DEVS engine and the hardware. Setting up and tuning an operational suite of hardware and software tools for experimentation with NPs can be tedious and complex. Moreover, in that context, implementing executives and low-level libraries can be a time-consuming and error prone. To cope with these difficulties we introduce a portable Virtual Lab for experimentation with ECD++ and the RadiSys ENP-2611 networking board [5] based on the IXP2400.

2. BACKGROUND

Different techniques have been proposed to achieve continuity and consistency of models when implemented in embedded systems. For example BIP [6] defines the components as a superposition of three layers: Behavior (a set of transitions), interactions (between transitions) and Priorities (to choose between transitions). BIP preserves properties during model composition and supports analysis and transformations between heterogeneous boundaries (timed/not timed, synchronous/asynchronous, event-triggered/data-triggered). Metropolis [7] is an environment for electronic system design that supports specification, simulation, formal analysis, and synthesis. It is based on meta-models with formal semantics, and it captures the design at a high level of abstraction, implementing different Models of Computation (MoC). Ptolemy II [8] allows hierarchical and structured modeling of heterogeneous systems using a specific MOC that provides data flow and control flow. ECSL (Embedded Systems Control Language) [9] supports the development of distributed controllers, including a specific domain environment for automotive systems (extending the Matlab family with capabilities for specification, verification, planning, performance analysis, etc.) SystemC and Esterel are system level languages used to simulate and run models that have benefitted from a growing industry adoption [9] SystemC represents hardware and software systems at different levels of abstraction, allowing choosing the desired level for each component. Esterel is used to synthesize hardware and software through a language based on reaction and high-level statements to handle concurrency. One of the most popular techniques is UML-RT, which provides an object-oriented methodology [11]. In [12] the authors propose a comparison between UML-RT and DEVS showing that despite Profile UML-RT specify time, planning, and performance using UML objects, they are not formally defined.

DEVS theory [2] provides sound semantics for representing structure, behavior, and time, leading to a well-defined and unambiguous MoC. However, DEVS is not intended for software design and development. Thus, it is essential to provide support for the evolution from DEVS models to equivalent software components that can operate on the complexity of embedded environments.

Different tools help in implementing the four functions required to build atomic DEVS models, and to declare the structural relationships required to build coupled DEVS models (see [14], [15], and [16], among others). In particular, ECD++ [17] is able to define and execute DEVS models in embedded environments with real-time capabilities, allowing users to develop Hardware-In-The-Loop (HIL) applications.

2.1. Network Processors

A Network Processor (NP) is a class of System-on-a-Chip (SoC) that combines the flexibility of generic purpose CPUs with high performance special-purpose circuits to transmit packets at line speed [18]. An NP combines heterogeneous devices into a single integrated circuit, including general-purpose processors, special-purpose microcontrollers, switch units, cryptographic units, memory controllers, etc. In our case, we used a 2.5 Gbps Intel IXP2400 processor [19]. This NP is structured in two levels: the Slow and the Fast Data Paths. The first one is a generic-purpose XScale processor (ARM V5TE, 600 MHz, 32 bit) called the *Core* processor, running RT Linux. The Fast Data Path consists of a cluster of 8 multithreaded RISC microcontrollers (600 MHz, 32 bits, called MicroEngines, ME), implementing a six-stage pipeline with a single clock cycle to complete. The IXP2400 allows the design of flexible reconfigurable rules in the Core, so that they can adapt dynamically without hindering the ability of the MEs to sustain their nominal packet processing [20]. Figure 1 (Left) shows the hardware architecture of this NP along with its associated software architecture (Right). Its design makes it possible to embed ECD++ in the Core, and use NP libraries to communicate ECD++ with the MEs. In this processor, network packets are transmitted via the Media Switch Fabric (MSF), which provides access to external managers of the network physical layer. The packets are received, processed and transmitted by Microcode running in the MEs. If needed, exception packets can be sent to the Core for special treatment. MEs do not have an operating system and are programmed in Assembler or MicroC, an adapted version of ANSI C.

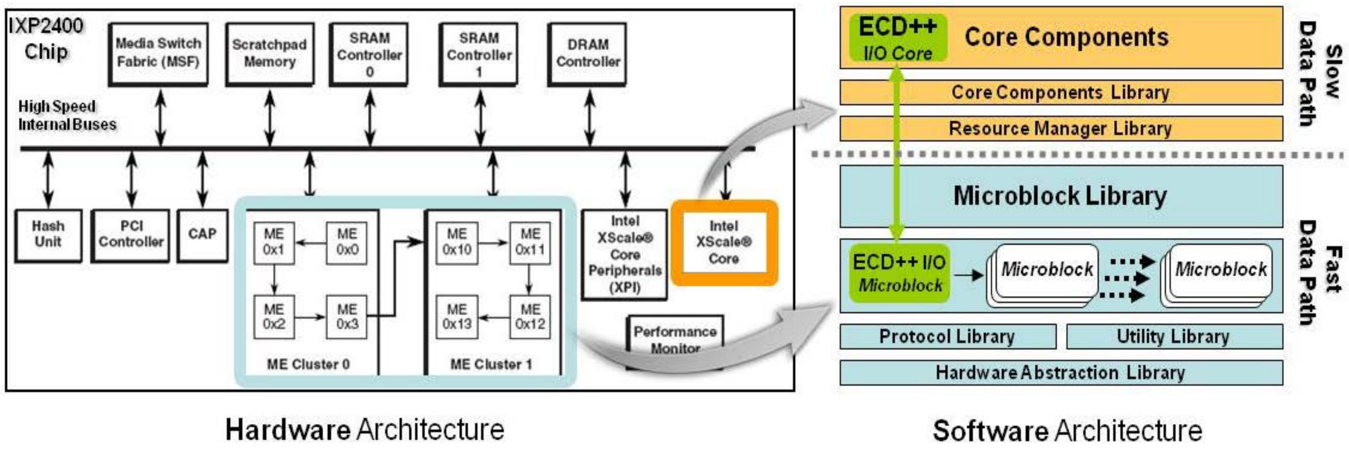

Figure 1. ECD++ embedded on an Intel IXP2400 NP

The MEs provide 8 hardware threads each, and use hardware signals to manage the context switching with zero latency. These characteristics, together with local low latency access registers, make it possible to handle packets at line speed. Internal timers provide a real-time accuracy of 16 clock cycles (~0.26 ns). Memory is organized hierarchically, and is shared between the ME and the Core: SRAM (8 Mb) and DRAM (256 Mb) are external (accessed both by the MEs and the Core) and a scratchpad memory (16 Kb) is used for fast signaling between ME threads and the Core (interrupts, data sharing, ring structures). Local Memory (2Kb per ME) provides low latency communication between MEs (not accessible by the Core). Special Next Neighbor (NN) registers can be accessed only by 2 adjacent MEs. Local memory access needs 1 clock cycle, the Scratchpad 60, SRAM 150, and DRAM 300 cycles. MEs operate much faster than the external memory, (instructions in 1 clock cycle), thus memory access is a source of blocking.

The orchestration of software tasks running at the Core with those running at the MEs is achieved by the standard Intel Internet Exchange Architecture (IXA) [21]. A network application based on IXA consists mainly of a pipeline of tasks applied to a stream of packets. Several tasks are distributed to MEs using different load balancing strategies.

3. ECD++ IN A NETWORK PROCESSOR

Figure 1 also shows the relationship between the NP hardware architecture and the IXA software architecture. At the Slow/Fast paths we can see three main libraries: *Core Components* (CC), *Resource Manager* (RM) and *MicroBlocks* (MB). CC is an Application Program Interface (API) that allows the creation of Linux kernel modules in the Core, which in turn uses the RM API to access the memory structures shared between the Core and the ME. As the same memory locations are referenced differently on each of them, the MEs use the MB API to access the shared memory using their own pointer structure. We mapped two components into these software levels: *ECD++I/O Core* is a *Core Component* invoked by ECD++ to communicate DEVS models with protocols at the Fast Data Path. *ECD++I/O microblock* is a piece of Microcode at the ME level that communicates with DEVS models at the Core level by "importing" variable shared at both processing levels.

From a DEVS modeler's perspective, the only requirement to communicate with MEs will be to agree with MicroBlock developers upon shared variable names. ECD++ will execute DEVS models at the Slow Data Path, and the subsystems representing "the network" will be replaced by ports to and from the real network hardware (i.e., the Fast Data Path with MEs running packet protocols).

4. IMPLEMENTATION

ECD++ was ported into the Core of the NP, and new libraries were developed for DEVS models to communicate with the traffic management layers, allowing embedded DEVS models to control high performance network packets.

4.1. Interface Library

The interface libraries allow any ECD++ atomic model to access variables in shared memory space with the MEs using the services *SetIXAVar()* and *GetIXAVar()*. Each atomic model interacts with MEs by meeting three requirements:

- ***Declaring*** one parameter of the atomic model as an *IXA Variable*. This is done in the coupled model (independent from the behavioral code), which specifies the structure of the coupled model and parameters for atomic models.

- ***Including*** the *ECD++IXA_Library.h* header in the atomic model's .cpp file. This provides the interfaces to invoke the *SetIXAVar()* and *GetIXAVar()* methods.

- ***Associating*** a block of Microcode to run on the ME space. The Microcode must import the same IXA Variable from the Core space.

The communication is resolved transparently through the *ECD++I/OLibrary* (Figure 2, Left). A *SW/HWMapper* DEVS atomic model uses the communication infrastructure to share variables with *ECD++Micro-block* Microcode.

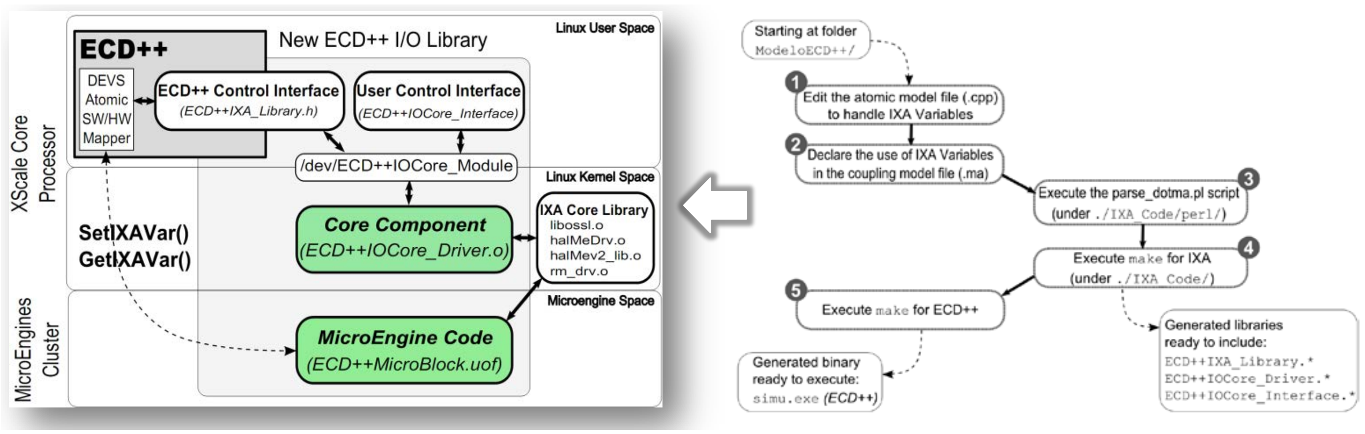

Figure 2. Left: ECD++ Input/Output libraries for IXA. Right: Automatic generation process.

Thus, Mapper models manage the access to the various shared external variables, encapsulating this functionality away from the rest of the models defining the dynamic specification of the system. The *ECD++IOCore_Interface* can be used from a console for monitoring or modifying shared variables, while ECD++ simultaneously operates on them.

4.2. Automatic Interface Generation

The IXA communication interfaces provide services to a set of user-defined IXA Variables. The libraries must be rebuilt whenever variables are added, removed or renamed. To do so, we designed a flexible mechanism for the generation of libraries, which automatically detects the IXA Variables declared by the modeler.

As seen in *Figure 2* (Right), there are five steps to produce libraries accessible for a DEVS atomic model. Steps 1 and 2 were discussed in the previous Section; they are needed to access IXA variables. These steps are the same for each ECD++ Mapper model. Steps 3 and 4 use Perl scripts to parse the coupled model file and generate libraries based on IXA variables declared. After the libraries have been generated, they are included to compile a new ECD++ simulator executive (Step 5). The binary obtained runs in the target processor (IXP2400 in our case). This mechanism automates the definition of Mapper models, hiding many low-level implementation details. This procedure assumes that, for each IXA Variable declared in a Mapper model, there will be a counterpart of Microcode (run in MEs) explicitly declaring the *same IXA Variable name* within the Core memory space (microcoding details are out the scope of this paper; it involves a development environment, tools, and code very specific to the underlying hardware; this information adds little value to modelers). The automatically generated interface code does not alter model behavior or the model coupling structure. Instead, it provides means for accessing low level, technology-specific memory positions from the code that implements the model logic. In this way, the modeler does not need to deal with platform-specific details. The overhead required to invoke the interfaces (from an ECD++ internal or external transition function) stay within the documented bounds for standard read/write operations between IXA Core Components and MEs [18,19]. This is an a priori knowledge a model developer must handle about the hardware environment.

4.3. Case study A: Basic Event Counter

We show the emulation of a traffic event monitor as a basic proof of concept for our methodology. An IXA Variable *EventCounter* will provide information about the number of certain traffic events occurring at the MEs, getting incremented by the MEs when the monitored events occur. The *ResetCommand* will allow resetting *EventCounter* to zero. Additionally, the *ResetCommand* code number may be consumed by the *ECD++MicroBlock* to perform additional actions. Figure 3 shows the coupled model definition for this example. The model includes a single atomic model named *eventCounter* (an instance of the *trafficEventCounter* atomic model type). Its outputs are sent through the port *counter* to the *counterStatus* coupled output port. *trafficEventCounter* uses two IXA Variables. The directive *useIXAVar[0]* indicates that they are shared with Microcode running on ME 0.

```
components : eventCounter@trafficEventCounter
out : counterStatus
Link : counter@eventCounter counterStatus
[eventCounter]
pollingPeriod : 00:00:02:000
counterLimit : 30000000
EventCounter : 0 %%useIXAVar[0]
ResetCommand : 7 %%useIXAVar[0]
```

Figure 3. IXA_EventCounter.ma Coupled Model

The atomic model *trafficEventCounter* (in Figure 5) queries the *EventCounter* IXA Variable (initialized to 0) in the ME, which counts the number of traffic events on each *pollingPeriod*. When *EventCounter* exceeds the *counterLimit*, the model must send a command through the IXA Variable *ResetCommand*. Command number 7 is interpreted by the Microcode as a request to reset *EventCounter*.

```
trafficEventCounter::trafficEventCounter(...):Atomic(name),counter(addOutputPort("counter")) {}

Model &trafficEventCounter::initFunction() {
    EventCounter = getParameter( description(),"EventCounter" )  ; // IXA Variable
    ResetCommand = getParameter( description(),"ResetCommand" ) ; // IXA Variable
    state = Sleeping;
    holdIn(active, pollingPeriod); // Time Advance ...} // End of Initialization

Model &trafficEventCounter::internalFunction(...){
 switch (state) {
     case Sleeping:   ...
        currentCounter = GetIXAVar("EventCounter");
        state = Notifying;
        holdIn(active, 0); // transmit immediately
     case Notifying:  ...
        if ( currentCounter > counterLimit )
           SetIXAVar("ResetCommand", strResetCommand);
        state = Sleeping;
        holdIn(active, pollingPeriod); ...
} // End of Internal Transition Function

Model &trafficEventCounter::outputFunction(...) {
   switch (state) {
         case Notifying:  // Transmit information observed
               sendOutput(msg.time(), counter, currentCounter); ...
} // End of Output Function
```

Figure 4. *trafficEventCounter* Atomic model

Figure 5 shows some details about this model. The *counter* port defined in the constructor will output values read from the ME. During model initialization, initial values for the IXA Variables are taken from the coupled model file in Figure 4. The model can be in two states: *Sleeping* or *Notifying*. The model starts *Sleeping*, and waits *pollingPeriod* seconds before sending the next query for *EventCounter*. When this time is consumed, the internal transition function triggers, reading the new value of the counter with `GetIXAVar("EventCounter")`, and changes the model state to *Notifying*. This transient state (lifetime=0) triggers the output function, which sends the last counter value through *eventCounter*; then the internal transition decides whether or not to send a reset command. If so, the command `SetIXAVar("ResetCommand",strResetCommand)` writes the IXA Variable *ResetCommand*, and the Microcode running in the ME reacts to this change. Finally, the model returns to the *Sleeping* state for *pollingPeriod* seconds, repeating the cycle. Figure 5 shows the execution results of the *IXA_Event Counter* model embedded in the ENP-2611.

Figure 5 shows 10 seconds of RT execution, in which the atomic model *eventCounter* detects twice a condition where the counter in the ME exceeds the *counterLimit* threshold. In those cases, the system sends the expected reset command, which is reflected in a drop of the counter below the threshold at the subsequent measurement period.

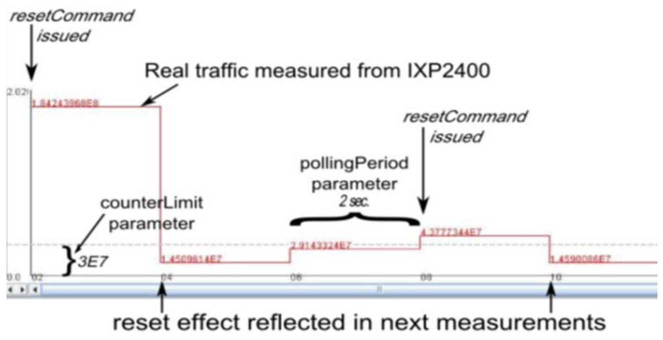

Figure 5. *IXA_EventCounter*: ECD++ in the IXP2400.

5. ADVANCED M&S TOOLS

CD++Builder [22] is an Eclipse plugin [5] for developing DEVS models integrating various existing tools within a common environment. It reduces the learning curve for new users and simplifies the definition and simulation of DEVS models. CD++Builder's graphical editors for coupled and atomic models enhance usability through a standard Eclipse GUI, and allow users to specify complex DEVS systems without programming, reusing existing libraries. Eclipse is a popular cross-platform environment with a familiar interface for users. It provides a framework designed to be extensible, making it easy to incorporate new functions within the same platform. Here, we extended CD++Builder in order to enable ECD++ DEVS models to be created and visualized using graphical notations, simplifying the building process for the embedded targets.

Figure 6 (Right) shows CD++Builder general features, including assistance for building embedded DEVS models for the IXA architecture. The graphical editor for DEVS coupled models (shown on the center pane) was extended in this work to support IXA Variable declarations. Atomic models that make use of IXA Variables can be represented graphically, and ECD++ simulation results that run on the embedded target can be visualized graphically. The compilation process now supports ECD++ integration into the NP, following the workflow in Figure 2 (Right), needed for generating customized IXA libraries. The ECD++ build process can run automatically, and the build process can be customized to configure the different parameters using a GUI. CD++Builder performs all the steps required to obtain an ECD++ binary executable on an IXA-compliant platform automatically. This reduces errors when running the ECD++ build tasks, expediting the experimentation process. The build flow runs integrated within the same environment used to design the models and visualize simulation results.

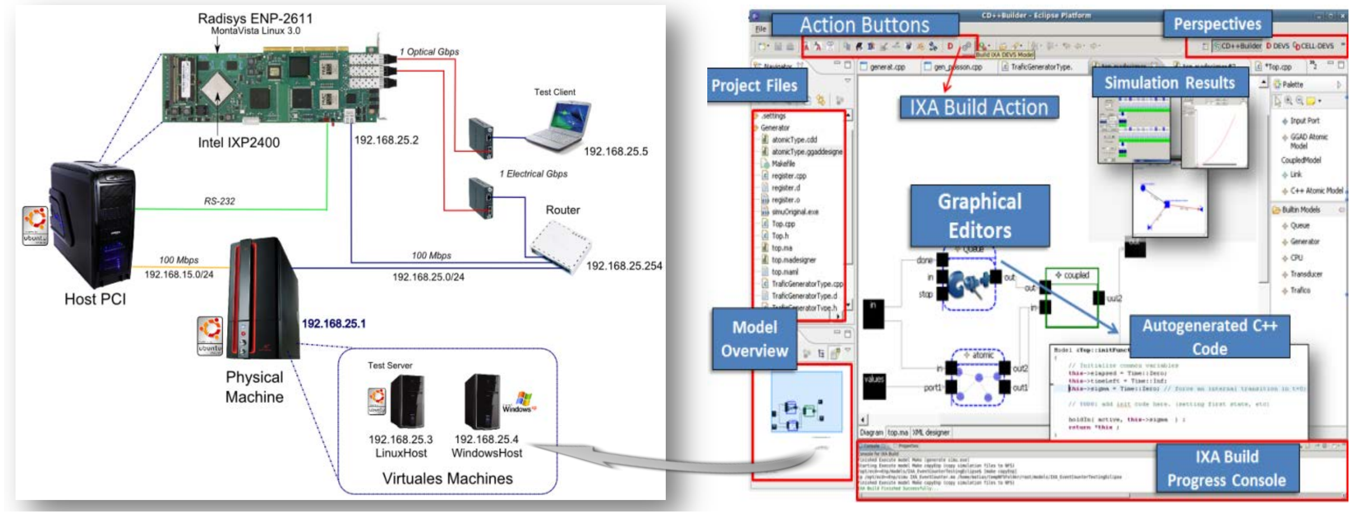

Figure 6. Left: Virtual Laboratory. Right: CD++Builder M&S Environment

5.1. Virtual Lab

Experimentation with the IXP2400 processor and the reference IXA architecture offers great power and flexibility to develop embedded applications for network control. However, installation and configuration of the many tools and libraries consists of tedious error-prone tasks, and experts' assistance, sometimes taking full working weeks. To solve these problems, we developed a reference virtual laboratory containing all the necessary infrastructure, tools and libraries set up in easily portable virtual machines. This provides an environment ready to create, test, and execute embedded DEVS models using the IXP2400 processor and the IXA reference architecture. Figure 6 (Left) shows a scheme of a testbed including the virtual laboratory combining logical and physical information. At the bottom of the figure, *LinuxHost* and *WindowsHost* represent virtual machines. In a separate machine (*Host PCI*) the RadiSys ENP-2611 board is connected in a PCI slot, administered via a dedicated Ethernet subnet and a terminal via RS232. If required, both physical machines can be collapsed into a single one.

To enable the execution of all the development tools, three operating systems are installed: Embedded Linux

(MontaVista) on the ENP-2611; Linux on *LinuxHost* (including the SDKs provided by Intel and RadiSys, ECD++, and the tools presented in this paper: the automatic generator of libraries for ECD++/IXA and the advanced CD++Builder GUI) and Windows on *WindowsHost* (needed to develop Microcode for the MEs using Developer Workbench, an advanced IDE provided by Intel).

CD++Builder in *LinuxHost* is used to design and develop control systems based on DEVS; the Developer Workbench on *WindowsHost* is used to create Microcode for the MEs. When both the DEVS models and the Microcode are ready, binary files are downloaded to the IXP2400 processor through automated scripts. After performing integrated real-time tests, results analysis can be conducted using the log files generated by ECD++ on embedded Linux, accessed via an NFS server mounted on the *LinuxHost*. This laboratory can be replicated, eliminating any preliminary effort before developing models for IXP2400 with CD++Builder.

5.2. Case Study B: Supervisory Control System

We now present a DEVS-based supervisory control system for Quality of Service (QoS) to monitor the traffic rate (TR) of packets, accepting dynamic policies for adapting its control rules. Reconfigurations occur at the Core of the NP, without risking the ability of the MEs to sustain nominal packet throughput. Depending on the policies and the state of TR, the system sends updated control actions to low-level algorithms at the MEs to enforce queue length management.

The information to and from MEs is delivered through Mapper models. All DEVS models impose their own overhead and also experience extra delays when interfacing with the MEs. Yet, this is the standard scenario for any code running at the Slow Data Path, regardless of which methodology it is developed with. In our methodology, the first step is to verify system behavior completely simulated in ECD++ (i.e., models run in the Core, not interacting with the MEs).

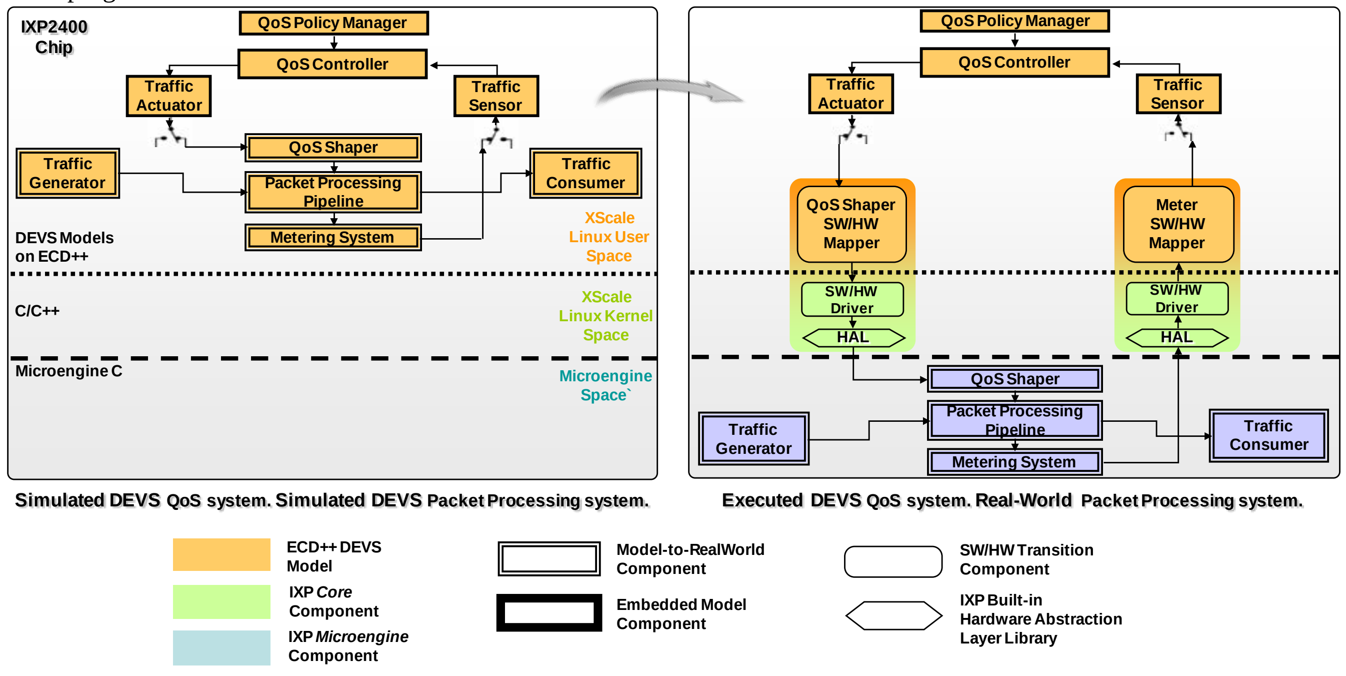

Figure 7. Left: Verification (embedded simulation). Right: Validation (embedded HIL execution)

Figure 7 (Left) shows a diagram of the control models. *Traffic Actuator* and *Traffic Sensor* are responsible for sending control commands and sense the TR, respectively. These models communicate with their counterparts *QoS Shaper* and *Metering System*, which emulate the packet processing hardware. Having verified the basic functionality of the QoS Controller model under a simplified scenario (using synthetic traffic generated by DEVS models) we proceed with the validation step under a realistic scenario: the *Controller*, *Actuator* and *Sensor* DEVS models are not changed, but now communicate with the real packet processors (MEs). In Figure 8 (Right) the actual hardware replaces the set of models that emulated its functions. This is possible thanks to the DEVS Mapper models (*QoS Shaper SW/HW* and *Meter SW/HW*), which operate as interfaces with MEs through the IXA Variables. A low-level algorithm running on the MEs should be able to apply Active Queue Management(AQM) techniques to ensure an average queue length, as indicated from the controller. In turn, the MEs make available updated information to the supervisory control about the number of packets transmitted at a given period. Thus, a QoS supervisory system designed with ECD++ can supervise an AQM controller by adjusting its set-point following a set of rules which may also vary according a QoS Policy Manager.

Figure 8 (Left) shows a snapshot of the QoS Control system modeled with CD++Builder. Figure 8 (Right) shows a DEVS Graph representing the control rules. The states reflect the condition of the packet processing system. The controller uses the *rateThreshold* to classify the intensity of traffic (High or Low). Initially, if *TR* > *rateThreshold*, the model transitions to state 1 and time T increases. If TR ≤ *rateThreshold*, the controller transitions to state 3. States combine the *TR level* (High or Low) and the amount of *uninterrupted time T* (Transient or Sustained) during which a given TR level is held. *sustainedThreshold* distinguishes between 2 values for the TR rate (Transient or Sustained).

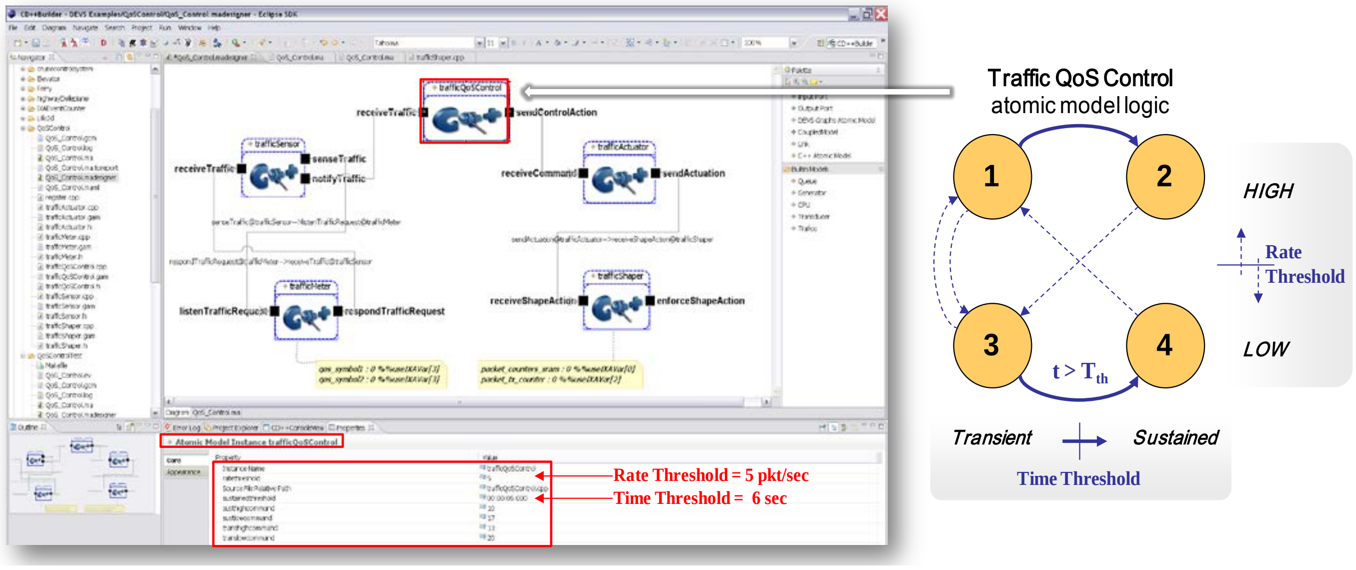

Figure 8. QoS_Control for IXP2400. Left: ECD++ Model within the CD++Builder GUI. Right: TrafficQoSControl logic.

When T crosses the *sustainedThreshold*, the model evolves from 1 to 2 depending on whether the system was in a high TR level or low TR level, respectively. When the traffic rate crosses the *rateThreshold*, T is reset and the state returns to 1 or 2. Both thresholds are parameters defined at the *trafficQoSControl* atomic model. Also the commands to be sent to AQM every time there is a state change in the state machine are initialized as parameters.

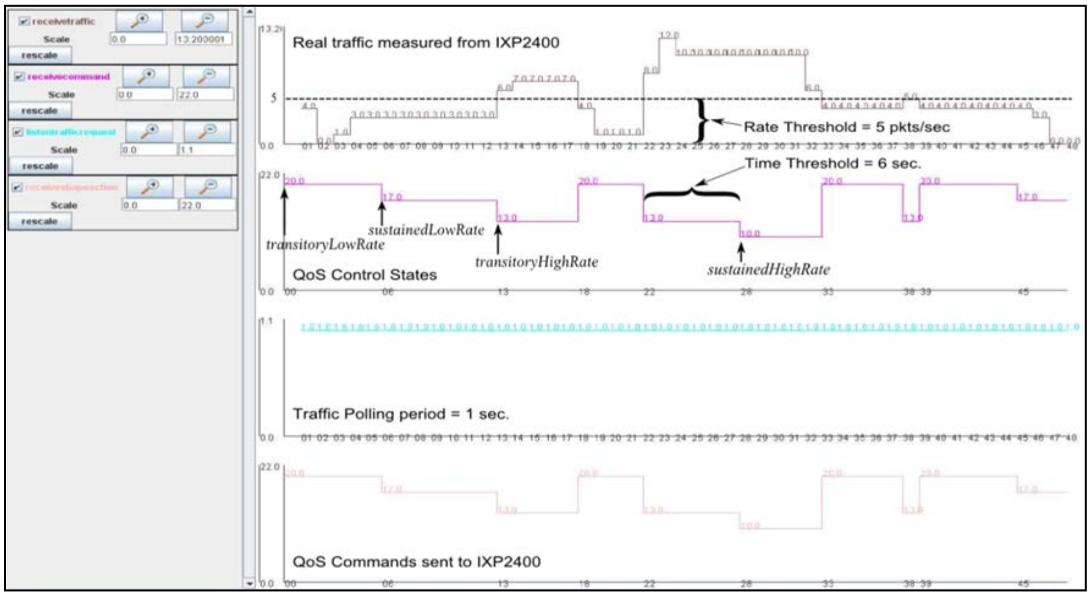

Figure 9. Real time simulation results

The *trafficActuator* block (responsible for sending these commands) ignores the difference between a standard DEVS model and a DEVS Mapper (capable of communicating with MEs). The Mapper role is accomplished by the *trafficShaper* model, declaring IXA Variables and invoking the *SetIXAVar()* service. Following the steps outlined in Section 4, an executable binary and ECD++/MEs communication libraries are generated for the *QoS_Control* system.

The validation step is done by placing a traffic generator in the LinuxHost server of the virtual laboratory (IP

192.168.25.3, see Figure 7, Left) generating packets for a Client machine (192.168.25.5). The route implies traffic passing through the ENP-2611, making the control system will react to real traffic measurement package. Experimental results are shown in *Figure 9*. The TR average level, measured by *trafficSensor/trafficMeter* every 1 second can be observed in the upper temporal sequence (*receiveTraffic* port). In the lower temporal sequence (*receiveShapeAction* port) we can validate the expected behavior for the state machine, which sends the correct sequence of commands to the ME. In turn, the temporal sequence for the *receiveCommand* port shows the states to which *trafficQoSControl* state machine transitions to. Therefore, we see that the supervisory control system reacts as desired to changes in the sensed variables.

6. CONCLUSIONS

We introduced new tools supporting a DEVS-based M&S methodology to implement real-time embedded network controllers. We obtained a simplified process that can produce final software products from DEVS models to embed them in the target hardware, and to execute in real-time interconnected with specialized traffic microcontrollers.

The ECD++ DEVS real-time simulator was embedded into the Intel IXP2400 hybrid network processor. By means of interface libraries DEVS models can interact with a cluster of in-chip microcontrollers for high performance packet handling. The continuity of DEVS models is guaranteed by encapsulating the interface with external hardware into special Mapper models, which invoke setter and getter functions to interact with low-level communication drivers.

CD++Builder environment provides visual modeling capabilities and automate the code generation of interface libraries for Mapper models, greatly enhancing the process of designing embedded systems for network control. This is one of many tools included in a portable Virtual Lab we built to facilitate experimentation with ECD++ and the RadiSys ENP-2611 networking board (based on the IXP2400 processor). Tools are pre-installed in virtual machines making it possible to easily reproduce the Lab, start developing controllers and deploy them for quick validation on hardware.

The examples studied show that final implementation and validation of DEVS-based network controllers can be carried out successfully; completely eliminating the need for adapting neither logic nor structure when evolving from standalone simulations (verification phase) to Hardware-In-The-Loop executions (validation phase).

The methodology promotes engineering solutions fully based on DEVS modeling and simulation.

7. REFERENCES

[1] Wainer, G., Glinsky, E., MacSween, P. "A Model-Driven Technique for Development of Embedded Systems Based on the DEVS Formalism". In Model-driven Software Development - Volume II of Research and Practice in Software Engineering. Springer-Verlag. 2005.

[2] Zeigler, B; Praehofer, H; Kim, T. 2000, "Theory of Modeling and Simulation", 2nd Ed. Academic Press.

[3] Saadawi, H., Wainer, G., Moallemi, M. "Principles of DEVS Models Verification for Real-Time Embedded Applications". In "Real-Time Simulation Technologies: Principles, Methodologies and Applications". Taylor and Francis. In Press. 2011.

[4] Wainer, G. "Discrete-Event Modeling and Simulation: A Practitioner's Approach". CRC Press, 2009.

[5] Budinsky, F., Brodsky, S.A., Merks, E. Eclipse modeling framework. Pearson Education, 2003.

[6] Bozga, M., Basu, A., Sifakis, J. "Modeling heterogeneous real-time components in BIP". In Proceedings of SEFM 2006, New York, NY. 2006.

[7] Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C. and Sangiovanni-Vincentelli, A. "Metropolis: An integrated electronic system design environment". IEEE Computer, 36(4):45–52, 2003.

[8] Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y. "Taming heterogeneity-the Ptolemy approach". Proceedings of the IEEE, 91(1):127–144, 2003.

[9] Balasubramanian, K., Gokhale, A., Karsai, G., Sztipanovits J., Neema, S. "Developing applications using model-driven design environments". Computer, 39(2):33–40, 2006.

[10] Brandt, J., Schneider, K. "How different are Esterel and SystemC". Forum on specification and Design Languages, FDL 2007, Barcelona, Spain. 2007.

[11] Selic, B. "Using UML for modeling complex real-time systems" Languages, Compilers and Tools for Embedded Systems. LNCS vol. 1474. pp. 250-262. Springer Verlag, 1998.

[12] Huang, D., Sarjoughian, H. "Software and simulation modeling for real-time software-intensive systems". Proceedings of DS-RT. Budapest, Hungary. 2004.

[13] Cellier, F., Kofman, E. (2006). Continuous System Simulation. Springer. New York.

[14] Sarjoughian, H, Zeigler, B. "DEVSJAVA: Basis for a DEVS-based collaborative M&S environment". Proc. of the International Conference on Web-based Modeling & Simulation, San Diego, CA. 1998.

[15] Bergero, F., Kofman, E. "PowerDEVS. A Tool for Hybrid System Modeling and Real Time". Simulation: Transactions of the Society of Modeling and Simulation International. 87(1-2) 113–132, 2010.

[16] Traoré, M. 2008, "SimStudio: a next generation modeling and simulation framework". Proceedings of SIMUTools 2008. Marseille, France.

[17] Yu, J., Wainer, G. "E-CD++: a tool for modeling embedded applications". In Proceedings of the 2007 SCS Summer Computer Simulation Conference. San Diego, CA. 2007.

[18] Comer, D. Network Systems Design Using Network Processors: Intel 2XXX Version. Prentice-Hall, Inc., 2005.

[19] Intel IXP2400 Network Processors. Intel Press, 2004.

[20] Gavrilovska, A., Kumar, S. and Schwan, K. "The execution of event-action rules on programmable network processors". Proceedings of OASIS 2004. Boston, MA. 2004.

[21] Carlson, B. Intel Internet Exchange Architecture and Applications: A Practical Guide to Intel's Network Processors. Intel Press, 2003.

[22] Bonaventura, M., Wainer, G., Castro, R. "Advanced IDE for modeling and simulation of discrete event systems". In Proc. of DEVS Symposium of Theory of Modeling and Simulation. Orlando, FL. 2010.