A Symbiotic Approach to Designing Cross-Layer QoS in Embedded Real-Time Systems

Florian Greff, Eric Dujardin, Arnaud Samama, Ye-Qiong Song, Laurent Ciarletta

▶ To cite this version:

Florian Greff, Eric Dujardin, Arnaud Samama, Ye-Qiong Song, Laurent Ciarletta. A Symbiotic Approach to Designing Cross-Layer QoS in Embedded Real-Time Systems. 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)", Jan 2016, Toulouse, France. hal-01242068

HAL Id: hal-01242068
https://inria.hal.science/hal-01242068

Submitted on 11 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Symbiotic Approach to Designing Cross-Layer QoS in Embedded Real-Time Systems

Florian Greff*†, Eric Dujardin*, Arnaud Samama*, Ye-Qiong Song† and Laurent Ciarletta†
*Thales Research & Technology - Palaiseau, France
{florian.greff — eric.dujardin — arnaud.samama}@thalesgroup.com
†LORIA Research Lab - University of Lorraine - Nancy, France
{ye-qiong.song — laurent.ciarletta}@loria.fr

***Abstract*—Nowadays there is an increasing need for embedded systems to support intensive computing while maintaining traditional hard real-time and fault-tolerant properties. Extending the principle of multi-core systems, we are exploring the use of distributed processing units interconnected via a high performance mesh network as a way of supporting distributed real-time applications. Fault-tolerance can then be ensured through dynamic allocation of both computing and communication resources. We postulate that enhancing QoS (Quality of Service) for real-time applications entails the development of a cross-layer support of high-level requirements, thus requiring a deep knowledge of the underlying networks. In this paper, we propose a new simulation/emulation/experimentation framework, ERICA, for designing such a feature. ERICA integrates both a network simulator and an actual hardware network to allow implementation and evaluation of different QoS-guaranteeing mechanisms. It also supports real-software-in-the-loop, i.e. running of real applications and middleware over these networks. Each component can evolve separately or together in a symbiotic manner, also making teamwork more flexible. We present in more detail our discrete-event simulation approach and the in-silicon implementation with which we cross-check our solutions in order to bring real performance aspects to our work. We also discuss the challenges of running real-software-in-the-loop in a real-time context, i.e. how to bridge it with a network simulator, and how to deal with time consistency.**

***Keywords:* mesh, network, real-time, QoS, simulation, RapidIO**

I. Introduction

We describe the background to our study, and explain our need to build a framework that allows real-software-in-the-loop to be run over both a simulated network and an in-silicon platform.

A. *Embedded Real-Time Systems: a Dynamic Approach*

Embedded real-time systems are part of many application domains. As a result of our background, we have closely observed their developments in transportation domains such as avionics, vetronics and UAVs. We have observed a growing tension between application needs and their execution constraints. We believe that these trends possibly apply to other domains.

On the one hand, we see an evolution of needs in terms of computing power and communication between applications. This is due to the changing qualities of sensors, whose data processing needs are increasing (for example in radar applications), and the emergence of new application categories

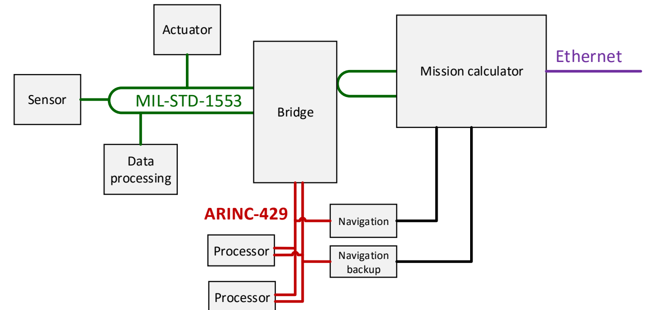

Fig. 1. Static system with real-time buses

such as multi-sensors. Interaction between applications and sensors is increasing, while features tend to be spread over several computing units. Communication architecture becomes more complex. Existing systems mainly process sensor data at high level (e.g. radar tracks). Multi-spectral imagery and cooperation between radar and electro-optical imagery would derive benefit from processing data coming directly from the sensors instead of the tracker. Image processing algorithms could be tuned in real-time in case of unconfirmed radar detection, in order to improve overall performance.

On the other hand, there are stringent constraints relating to hard real-time, criticality (endangerment of the mission or human lives in case of malfunction) as well as material constraints such as size, weight and power (SWaP). When data exchanges between components have hard real-time constraints, network access is typically budgeted (e.g. with TDMA, or static virtual channels in ARINC-664p7) and real-time buses such as MIL-STD-1553B or ARINC-429 are used, as illustrated in Figure 1. Using these buses brings several limitations in terms of throughput and dynamicity (i.e. online dynamic reconfiguration).

We believe that mesh networking of the components of such systems (Figure 2) would reconcile their constraints with the new needs of applications. The plurality of communication paths should result in increased flexibility, resilience, scalability and load balancing characteristics. To improve overall system resilience, it must be possible to redeploy a processing function running on a faulty computing unit to a non-faulty one or operate a graceful Quality of Service (QoS) degradation (Figure 3).

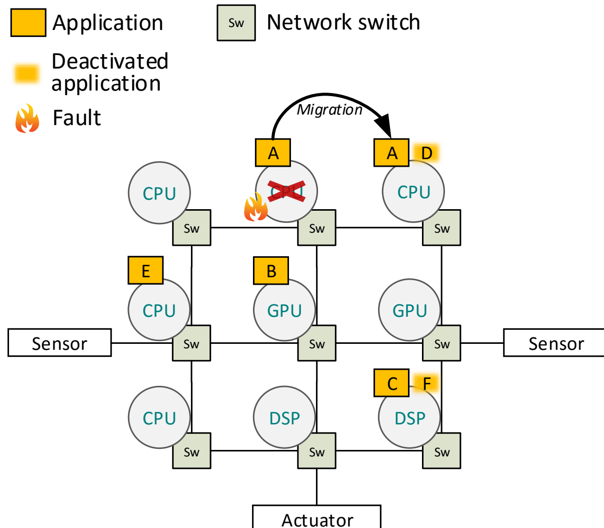

Fig. 2. Dynamic mesh networking of components and applications

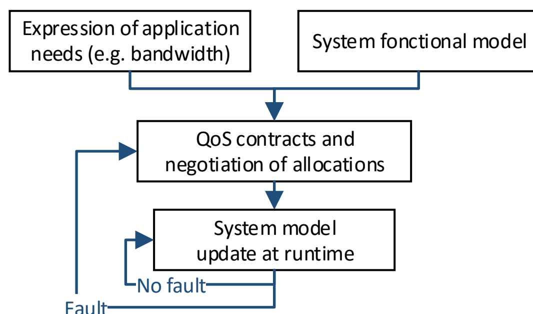

Fig. 3. Basic working of the dynamic system

For this project, we have chosen, for the time being, RapidIO® [1] [2] as the mesh networking technology. RapidIO was initially designed for embedded systems. It is used in most 3G/4G base stations, medical imagery and high performance computing. Its goals are a very high throughput and guaranteed low latency while providing reliable communication. Low latency is achieved thanks to Control Symbols that can be embedded within packets. They can be seen as small protocol headers used for flow control. The ability to send these Control Symbols even during a packet transmission allows very fast error recovery and thus decreases average end-to-end latency. RapidIO also allows assigning of priorities or virtual channels to flows. However, the way these priorities and virtual channels are used for switching is to be defined by the user. End-to-end flow control can be achieved in multiple ways (credit-based, XON/XOFF, etc.). RapidIO also specifies the definition of register fields for the endpoints and switches. They are used for switch configuration and monitoring. Finally, the standard specifies a clock synchronization protocol, a network discovery algorithm and a very fast fault recovery mechanism. Some additional information is given in Table I.

TABLE I. SOME CHARACTERISTICS OF RAPIDIO

Frame size	1-280 Bytes
Payload size	1-256 Bytes
Maximum bandwidth for four lanes	40 Gbps
Network layer	Cut-through routing support Store-and-forward
Routing protocol	Implementation dependent

B. *Expressing QoS Needs at Network Level*

Achieving this dynamicity in a reliable way requires knowledge of the deadline requirements between the different components. We assume that a good way of dealing with these requirements is a contract-based approach [3] [4]. Contracts allow identification, at runtime, of which network and processing resources are allocated to each application. Our goal is to make this easier by providing cross-layer QoS support from a middleware down to the hardware network switches, thanks to an adaptation layer.

To specify the application QoS (e.g. relative deadline), we consider the use of a Data Distribution Service (DDS) middleware [5]. DDS is a machine-to-machine middleware standard from the Object Management Group, based on the *publish-subscribe* paradigm. It aims to provide a scalable, high-performance and dependable real-time data exchange API. It addresses the needs of various domains such as aeronautics, defense, big data and telecommunications. Publish-subscribe communication seems to fit our needs well, since we aim to support data-intensive applications that can be distributed across multiple nodes (radar for instance). Although DDS provides an interesting way of allowing the QoS requirements specification, there is no underlying mechanism for providing any real-time QoS guarantee, to the best of our knowledge. DDS is indeed designed to run over any type of network and limits itself to indicating the deadline meeting ratio, for instance.

For the implementation of DDS, we have selected Vortex OpenSplice. It was initially developed by Thales Naval Nederland and is now under the auspices of PrismTech as Vortex OpenSplice [6].

To provide full real-time cross-layer QoS support, a complete understanding of network-induced delays is required: buffering, arbitration logic, propagation delay, error recovery based on probabilistic model, contention, etc. We can then design an adaptation layer within the middleware that will translate QoS parameters expressed at application level into what is needed at network level to enforce them (e.g. via packet headers contents).

These assumptions lead us to two major questions:

1) How do we translate QoS defined at contract level into QoS at network level?

2) How can we make use of network parameters (routing, output port packet scheduling, flow control) to reach the desired QoS?

To answer these questions, we first need to build a tool that allows us to implement and test those QoS translation and network-level QoS handling mechanisms. The goal of this paper is to present our symbiotic approach to achieving this goal. This will, in turn, allow us to further answer the above two questions in our future work.

C. The Need for Experimentation

We aim to assess the network properties required to achieve real-time QoS cross-layer support within a mesh network (i.e. with dynamic tasks and network resource re-allocations). For this purpose, we have built a test platform based on the open Serial RapidIO standard, of which an open-source VHDL implementation is available. Having an in-silicon platform is ideal for assessing our final solutions in a real context. However, the complexity of the implementation and the fact that the hardware architecture cannot be changed easily make it difficult to test a lot of algorithms in order to compare them. For that reason, we consider it very useful to work on the key features of cross-layer QoS support (switching, communications scheduling, graceful degradation, etc.) in the context of a network simulator, before assessing them on the real platform.

On the one hand, building the whole system including application code, middleware, OS, CPU/GPU and RapidIO mesh network results in a total experimental approach. On the other hand, simulating all these components would require a great deal of abstraction (at the expense of losing some features) to obtain simulation models of them, whilst it is well-known that modelling applications/middleware/OS/CPU is a difficult task in general.

As a result, we propose to keep using the same upper layers (application code, middleware, OS, CPU/GPU) while having the choice of working on the simulated network or the real one. This allows us to reuse the same set of applications for both networks, thus making implementation and evaluation of cross-layer QoS support and fault-tolerance features easier. It also makes simulation assessment more relevant. Indeed, the only layer to be switched is the network one, so we know that applications and middleware which we would otherwise have simulated do not cause bugs or bad behaviour that we would then have to fix. To the best of our knowledge, there is no experimentation framework able to support such an interaction between real software including middleware, simulated networks and in-silicon ones.

The main contribution of this paper is to propose our symbiotic simulation/emulation/experimentation architecture, ERICA[1], that we use to design real-time QoS support. Applications and DDS middleware can both run above real-world networks and network simulators, thanks to an adaptation layer that we have defined and implemented. We also explain how we have developed a coarse-grain discrete-event simulator achieving the dual goal of accuracy for computing transmission time, and efficiency for running along with operational code. The Ptolemy II modeling framework [7] was used for this purpose. The simulator was tuned according to the run-time platform and then used to help design the latter. Finally, we show the benefit of such a symbiotic approach.

To achieve the interaction between application execution and simulator run, two technical challenges must be solved. First, how to link one application execution to a simulator? Second, how to synchronize the time of the application execution to that of the simulator? This paper is dedicated to answering the first question. We will address the second one in further research; it is sufficient for now to note that this synchronization is not related to real-world time, but to the consistency between the latencies perceived by real software and those emulated by the simulator.

The remainder of the paper is organized as follows. Section II reviews some existing multi-simulation approaches that combine either simulation with hardware-in-the-loop, or two or more different simulators, or one emulator and one simulator. Section III describes the ERICA architecture. Section IV illustrates the results of our approach. Section V discusses the current state of ERICA and presents our ongoing work. We then provide our conclusions in section VI.

II. Related Work on Multi-Simulation Approaches

The idea of using a simulator to emulate part of a more complex system has been highlighted by parallel and distributed simulation (PADS)[4] [8]. PADS addresses the need to distribute a comprehensive simulation, in order to obtain speed-up – by parallelization of computation tasks –, reuse existing simulation components or utilise geographically distributed design environments (e.g. for military training).

High Level Architecture (HLA) [9] is the leading standard for bridging heterogeneous components into an overall experimentation architecture [10] [11]. It is driven by the need to couple existing simulators or *live players* (i.e. users who can interact with the simulation at runtime) without having to design a complete architecture from scratch [12] [8]. It allows hardware to be brought into the loop thanks to live players interfaces. It also specifies a way of achieving time synchronization while federated components may have heterogeneous timing constraints (conservative, optimistic, etc.) [10] [13]. Ptolemy II extensions for its integration within an HLA federation have been designed [14]. However, HLA appears to be very heavy, complex and time consuming when used for designing an overall architecture [11]. Moreover, there is no standard use pattern for it, leading to the need for additional interoperability packages [15] [16].

MECSYCO [17] combines the Discrete EVent System Specification (DEVS) [18] formalism and multi-agent concept to enable the integration of heterogeneous formalisms and manage simulator interoperability. It is notably used for smart grids simulation [19], with existing simulators, although Ptolemy II is not supported yet. However, it does not allow hardware to be brought into the loop and is thus not suitable for our project, since we want to use an in-silicon platform to assess the network solutions at different steps of their design.

COOJA [20] is a wireless sensor network simulator. It includes a microcontroller emulator (e.g. MSPSim for emulating

[1]ERICA stands for ExpeRImenting Cross-layer QoS for Autonomic systems

MSP430 MPU) and a simple radio channel simulator, so that the same sensor source code and OS can run indifferently in COOJA or in a mote (e.g. a MSP430-based TelosB mote). This approach is interesting but slightly different to ours, which runs the application and middleware code over actual hardware machines and a simulated network.

Symbiotic simulation is a more general paradigm in which a simulation and physical systems are closely associated with each other [21]. It is mostly used in the context of ”what-if” analysis, i.e. runtime control of the hardware through a simulator that analyzes several scenarios. It also brings the idea that data from physical sensors can improve the accuracy of the simulation. This idea is central to our approach. Indeed, we advocate that data captured from real systems are key both to improving the accuracy of simulation, and to instrumenting the simulation while searching for an appropriate system design.

III. The ERICA Framework: Simulation/Emulation/Experimentation

A. Overall Description

Since we have to work at two different levels (i.e. middleware and network), we need an experimentation architecture and associated tools that allow us to:

- Test our DDS adaptation layer over a well-controlled network
- Design and test network features (e.g. routing or scheduling algorithms) that will be used for QoS enforcement
- Validate our implementation with respect to the expected behaviour of network components such as switches

We have defined a two-layer approach for this purpose, ERICA (Figure 4):

- The high layer is made of real benchmarking applications and middleware (DDS)
- The low layer is the network layer, either real or simulated

The adaptation layer, which will be described below in more detail, allows the same applications and middleware to be run over different kind of networks or simulators at the lower layer.

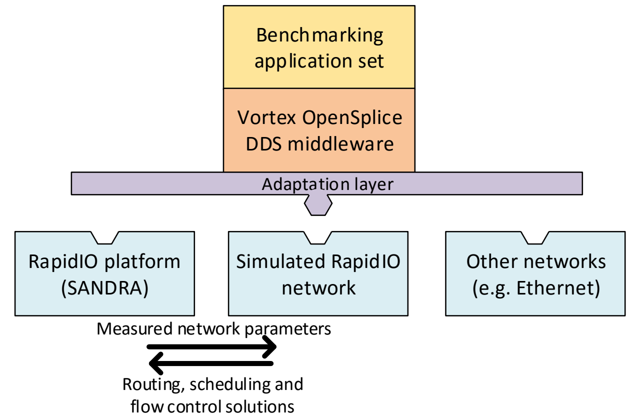

Fig. 4. ERICA: a two-layer experimentation architecture

The simulated network is used:

- To model a RapidIO network in order to test out routing, scheduling and flow control algorithms (see C)
- To model simple networks in order to test out the way our system translates QoS parameters expressed at the DDS level into network configuration or packet header contents (see D)

The in-silicon platform (see B), used as a reference, gives us some reference values to inject into the simulator (e.g. link latencies), so that we can experiment our features (reconfiguration, routing, etc.) in a representative environment (Figure 4). We then compare its behaviour with that of the simulator, in order to validate both the simulation model and the implementation (see section IV for more details).

B. Experimentation: a RapidIO Platform – SANDRA

Our in-silicon RapidIO experimental platform, *SANDRA*[2], is based on an open-source version available from Open Core [22]. This open-source project, founded by Bombardier® [23] in 2013, provides the physical and logical layers for Serial RapidIO (SRIO), but also one implementation of a SRIO switch.

In order to have a fully observable network adapter, we used this foundation on a Zynq ZC7045 board. The Xilinx® Zynq® programmable SoC [24] [25] [26] is made of an FPGA and a dual-core ARM® [27] Cortex-A9 CPU on the same die. The FPGA and the ARM are tightly connected through several buses, notably 4 high-performance AMBA AXI® [28] slave interfaces and 4 general purpose AXI interfaces (2 slaves, 2 masters). We are using the FPGA to implement our network adapter. It is made of a 5-port SRIO switch. One of the ports is connected to a SRIO-to-AXI bridge, and then connected to one of the High Performance AXI ports of the ARM CPU. We are running a Linux partition on top of a hypervisor. This partition contains our test application running on top of DDS. An illustration of the overall platform is given in Figure 5.

[2]SANDRA stands for Self-Adaptive Network for Distributed and Reconfigurable Applications

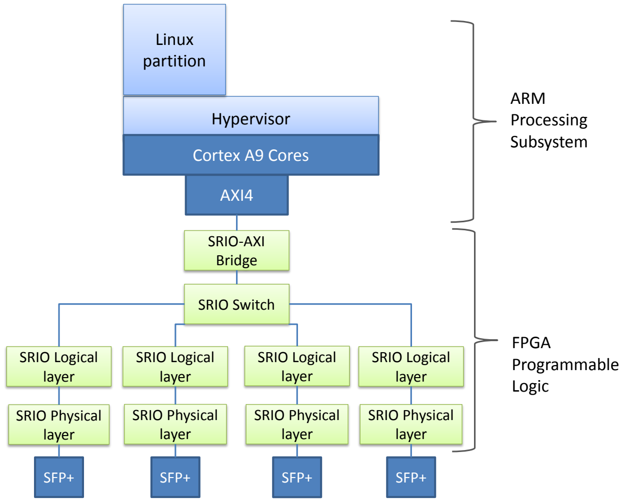

Fig. 5. SANDRA architecture

If we consider a typical network adapter or switch, the internal logic and arbitration is not known. In our case, thanks to the VHDL source availability, we are able to understand and investigate any behaviour.

The FPGA design tools provide precise timing measurements. First, they can run cycle-accurate simulations of the design. We can feed them with test patterns such as single packets, single flows, then simultaneous packet arrivals, link errors, arbitration logic changes, etc. The resulting time measurements are key to testing our design, to pinpointing the root cause of possible erroneous behaviour observed at macroscopic level (e.g. unexpected latencies), and reliably estimating the latency associated with each logical block. These design tools also allow insertion of probes to record true in-silicon measurements at runtime, so as to confirm the results of this cycle-accurate simulation.

C. Network Simulation: an Example Using Ptolemy II

1) Architecture of the Simulation Model: We are designing this simulator using a discrete-event model. This has been shown as appropriate for network simulation [29] [30]. We are simulating the functions of SRIO adapters, and our references are the SRIO standard [1] and the VHDL implementation, especially the use of the same buffering along the transmission path. This work has been done on the Ptolemy II modelling framework, developed at the University of California Berkeley [7].

The goal was to keep our design modular enough to avoid complexity. While basic elements like FIFOs and buses – that we are not willing to work on once the model is validated – are modelled in a strong though more complex way (i.e. graphic), Python scripts are used for simulating routing and scheduling elements. Such a programming language brings more expressivity and flexibility for the key features related to cross-layer QoS support, which we need to change and test frequently, while the rest of the model remains as visually clear as possible.

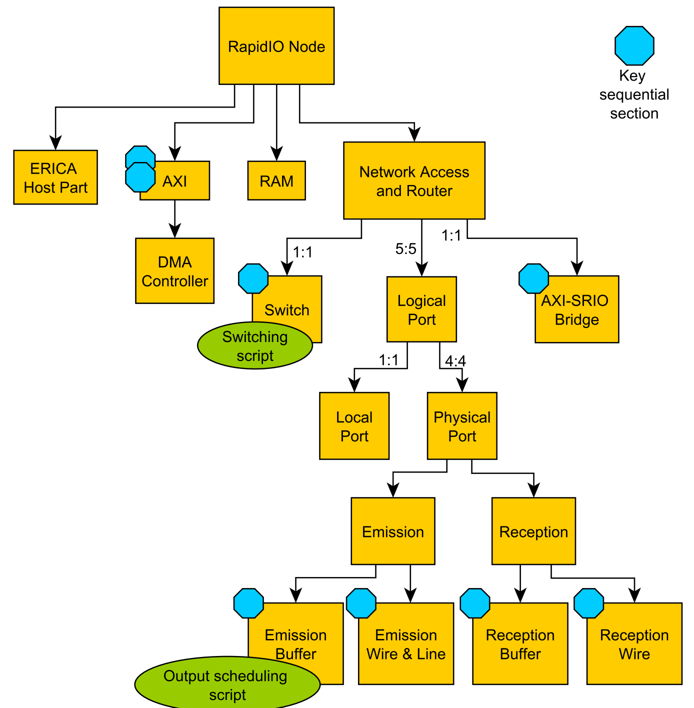

Fig. 6. Composition hierarchy of the RapidIO model

In order to avoid pointless complexity, our RapidIO model focuses on the fonctions which have a significant impact on determinism, flow-control and routing. Indeed, we are not aiming at very precise measurements, but only at transmission times accurate enough to let us work on cross-layer QoS support in a representative environment. As we will detail in section IV, measurements from the platform allow us to assess this accuracy.

We identified *key sequential sections* within RapidIO nodes, in which the individual latencies of logical blocks can be gathered into a *global latency*. This applies when the transition across them is deterministic, i.e. there is no protocol to model between the beginning and the end of the section. Obviously we retain only the sections whose global latency is significant enough to be considered. These are:

- RAM to switch via AXI Bus and Bridge, for read actions
- Switching delay
- Emission/reception buffers to switch and vice versa
- Emission/reception buffers to line and vice versa
- Switch to AXI-SRIO Bridge, for write actions
- AXI-SRIO Bridge to RAM via AXI Bus, for write actions

The composition hierarchy of our model is shown in Figure 6. Note that it is a logical view of the platform, not a direct representation of it.

Once the simulator behaviour has been validated with respect to the SRIO standard, latencies are tuned thanks to the in-silicon platform and its cycle-accurate simulator. For

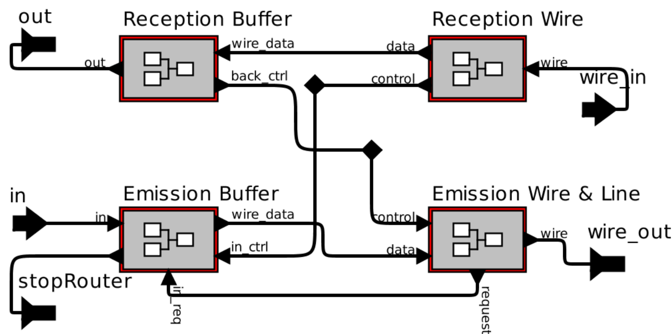

Fig. 7. Structure of the Port actor

each sequential section, we measure the traversal time for a set of packet sizes. We then identify a formula that computes the traversal time of the sequential section depending on the packet size, close enough to the measurements. More details about this process are given in section IV.

To illustrate our approach, we focus on the `Physical Port` actor. This actor is composed of four components, since it distinguishes between, on the one hand, the inbound and outbound traffics, and, on the other hand, the wire level and the packet buffering level, as depicted in Figure 7:

- At wire level, the `Emission Wire & Line` and `Receive Wire` actors handle sending of packets and Control Symbols. In RapidIO, Control Symbols may interrupt packets. For this reason, we send packets as two events (packet beginning, packet end), while Control Symbols can be sent in between
- At packet buffering level, the `Emission Buffer` and `Reception Buffer` handle CRC checking, the related ACK/Retry mechanism, and output queuing

Output queuing is defined as a Python script. It supports multiple priority levels, manages a dedicated queue of packets waiting for acknowledgement, and coordinates with the central switch for flow control. In the current version, we make simple choices:

- Priority levels are strict: there is no fair share discipline
- Each packet is ACK'ed individually: there is no group acknowledgement
- Packets to be re-emitted are put at the tail of the emission queue, like incoming packets
- When a queue is full, the port stops the main switch until a packet successfully leaves the queue

These choices can be flexibly changed in the Python script to search for better solutions. For example, priority queuing relies on per-priority queues; hence a filled-up, low-priority queue may block the main switch while taking time to drain. Improving that, while keeping the complexity low for efficient in-silicon implementation, is part of the challenges that our approach will help to address.

D. Adaptation Layer: Allowing Real Software in the Loop

The ERICA adaptation layer (see Figure 4) is composed of the following elements:

- The adaptation module integrated into DDS, which adapts its behaviour according to the underlying network
- A set of Ptolemy II extensions, namely:
  - `EricaAppPart`, which listens to packets from DDS, tags the data with their metadata – one of them being the application ID – then sends them to `ERICA Mapper` (and vice versa)
  - `ERICA Mapper`, making the bridge between simulated RapidIO nodes and applications
  - `EricaHostPart`, filtering packets from `ERICA Mapper` which relate to the node

We will now describe these components in more detail, and provide illustrations.

1) DDS Adaptation Module: The adaptation module we introduced in section III-A is the cornerstone of the ERICA architecture. It enables the use of DDS middleware over either a Ptolemy II simulated network or the SANDRA platform. It could eventually be extended to work over other networks – note that DDS already works over UDP/IP through its base implementation.

At first, working with Ptolemy II will allow us to test out how the DDS QoS parameters are being translated on networks of increasing complexity. Since the locations of the applications on the network are important and must be taken into account, it is very useful to be able to create different topologies over which our adaptation layer will be tested out. Ultimately, this will allow us to stress our fully simulated network with a real middleware behaviour that could neither be accurately nor simply simulated. As we explained in the introduction to this paper, bridging the simulated and real-world networks under a common application layer also makes it easier to focus on them, whatever happens above the adaptation module.

We have integrated the adaptation module within Vortex OpenSplice DDS, at the step where DDS packets are about to be sent out through a UDP socket. At this step, the module is basically intercepting the data buffers, IP adresses and ports according to the underlying network. If the underlying network is Ethernet, the base behaviour is retained, i.e. sending the packet through the socket. If the underlying network is the SANDRA platform (i.e. an in-silicon RapidIO node), addresses are translated to RapidIO addresses and the sending function from the SANDRA API is called. If working over a Ptolemy II simulated RapidIO network, the DDS adaptation module wraps the data with associated metadata (application ID, node ID, destination ID) and sends them to a UDP listener at the Ptolemy II level.

2) Ptolemy II Extensions for ERICA: The main idea is to use Ptolemy II publish-subscribe actors to map applications to their corresponding node in the simulated network. Publish-subscribe actors are useful for connecting applications to their corresponding nodes, without having to physically draw any

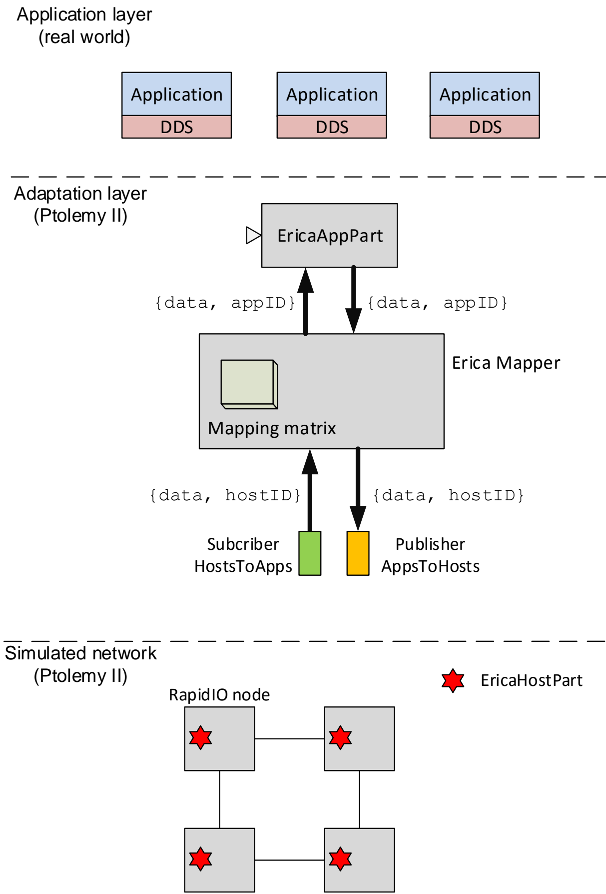

Fig. 8. Overall architecture of the Ptolemy II adaptation layer

actor link. This allows us to separate the application and network layers and centralize the mapping into a unique matrix, thanks to the design described in Figure 8.

All applications are connected to all nodes through two pub-sub channels that go from RapidIO nodes to `ERICA Mapper` and vice versa.

The `EricaAppPart` actor (Figure 9) listens to packets from DDS, parses the metadata of incoming packets, and then emulates the sending of the corresponding RapidIO packet from the right node, via `ERICA Mapper`. Conversely, when receiving data from the simulated network, it rebuilds the IP packet expected by the upper DDS layers and forwards it via UDP to the application. This leaves the `EricaAppPart` fully transparent from the DDS point of view. `EricaAppPart` also contains an input multiport into which simulated applications can be plugged, allowing mixing of real and simulated applications.

The `ERICA Mapper` actor is composed of a mapping matrix and a script:

- Downstream, it tags the data from the applications with the target host ID and publishes it to the nodes. Although each node then receives these data, they all discard them, except for the node with the corresponding host ID. This filtering is done thanks to

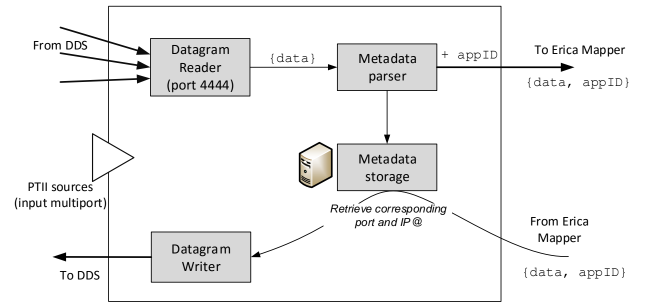

Fig. 9. EricaAppPart model

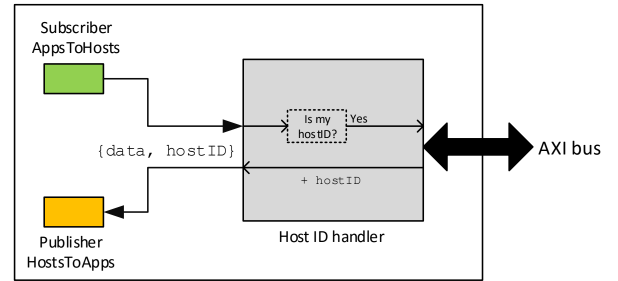

Fig. 10. EricaHostPart model

  the `EricaHostPart` actor (see Figure 10).
- Upstream, the RapidIO node tags the data with his host ID and publishes it to `ERICA Mapper`, thanks to `EricaHostPart`. The `ERICA Mapper` component then looks into the mapping matrix and forwards the data to the right application, by tagging them with the target application ID.

Using this technique, the application set and the network are almost separated: changing the underlying topology or the application set only involves editing the mapping matrix without drawing or deleting any actor link.

3) Wrap-up: With our architecture, the application and DDS layers may address either UDP/IP or our network, simulated or in-silicon. Conversely, either these applications, or simulated ones (based on Ptolemy II source actors) may address our simulator. However, although this architecture does work with one application and one node, it does not work with multiple applications and nodes, for a simple reason: Ptolemy II does not allow the creation of more than one `Publisher` for a given channel, because it would make the model non-deterministic. We thus have to adapt this base design, by creating dedicated channels per node (Figure 11). We are willing to find a better solution than having to manage so many channels, however. Except for this issue, Ptolemy II has shown very acceptable performances for the simulation of RapidIO with our model, given the scenarios we fed it with.

To summarize, our architecture allows real and simulated components to work together, and to be switched in a flexible way. This is required in order to experiment with routing

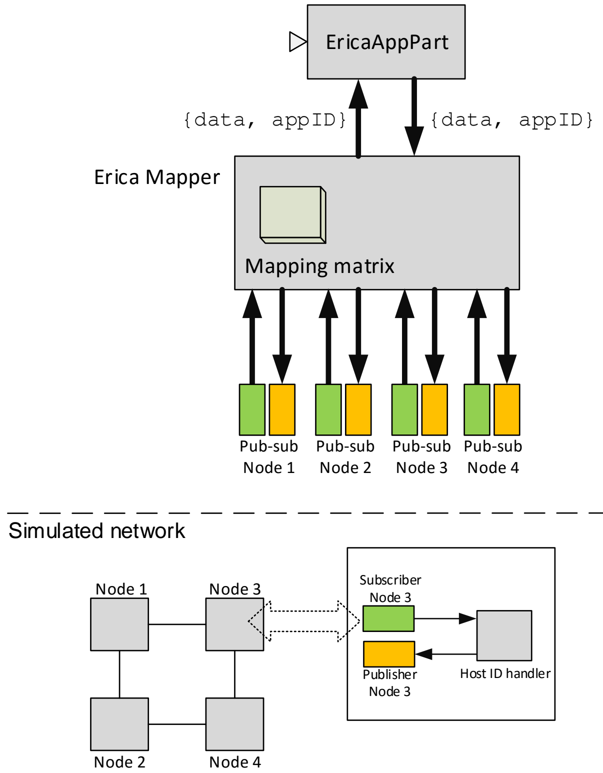

Fig. 11. Current architecture of the Ptolemy II adaptation layer

policies on a step-by-step basis, from full simulation to a real application on a physical network.

IV. Experiments and Results

Once the simulator behaviour has been validated with respect to the RapidIO specification, we have to tune the latencies and verify that end-to-end latencies are consistent with those measured on the platform. For this purpose, we firstly use simulated applications (i.e. Ptolemy II source actors) in order to ensure very simple, harnessed communications.

A. First Step: Overall Checking of the Model

The first scenario is the basic point-to-point sending of a 4-byte payload, RAM-to-RAM. The goal is to validate our *sequential sections* and make sure that the overall simulation architecture is accurate enough with respect to the platform, for this simple scenario. We measure, on the platform, the values of latencies for each sequential section, then inject them into the simulation model, at the places shown in Figure 6. We then measure the end-to-end (i.e. RAM-to-RAM) latency on the simulator and the platform in order to compare them. Although we could just calculate this latency – by summing values – to make sure that our theorical model is good compared to the platform, measuring it from the simulator allows us to assess the actual model in the same process. Measurements have been performed three times, and average values are given in Table II.

As we can see, there is a huge gap between end-to-end latencies, although each latency section has been tuned with

TABLE II. Measured latencies for sending of a simple 4-byte payload

Sequential section	Latency (ns)
AXI read	296 ns
Switching delay	56 ns
Emission Buffer	124 ns
Emission Wire	285 ns
Reception Wire	355 ns
Reception Buffer	124 ns
Switching delay (2)	56 ns
AXI-SRIO Bridge	168 ns
AXI write	226 ns
Σ	1,690 ns
RAM-to-RAM on PTII	**1,690 ns**
RAM-to-RAM on SANDRA	**5,100 ns**
Δ	**67%**

TABLE III. Measured end-to-end latencies after the implementation bug fix

RAM-to-RAM on PTII	1,690 ns
RAM-to-RAM on SANDRA	1,625 ns
Δ	**4%**

exact values. This gap is unacceptable because it would mean that the majority of significant latencies has been neglected, making our simulation not sufficiently reliable. This issue has two potential causes: either we have to take other latencies into account, that we did not think of; or there is an issue with the platform implementation, leading to the appearance of an unexpected latency. After investigating this issue, we find that a data width conversion FIFO (32 bits to 64 bits) is the source of the problem. This 64-bit bus is used to feed 4 transceivers when running in 4x mode with 16 bits of data per clock cycle. The FIFO always waited to be filled to 64 bits before forwarding the data. In our case, since there was only one packet to be sent, the End-of-Packet Control Symbol was stuck inside it. This would have happened each time the transferred data were not aligned to a 64-bit boundary. After having removed this latency, the measured end-to-end latencies are those given in Table III.

For this scenario, we showed that our model is accurate enough in terms of latencies, even if the end-to-end value from the simulator is slightly higher than that measured on the platform. This is due to measurement uncertainties, notably for RAM latencies, whose values may significantly vary. As previously explained, we are not aiming to compute Worst Case Transmission Time, but only a sufficiently accurate transmission time.

This first scenario shows the benefit of the symbiotic approach of ERICA, which is to cross-check values from the simulator and the in-silicon platform. This approach helped us to find a major issue within the implementation that we would probably not easily have found otherwise, considering

TABLE IV. MEASURED LATENCIES FOR THE EMISSION BUFFER SEQUENTIAL SECTION AND DEDUCED AFFINE FUNCTION

Packet size (bytes)	Latency (ns)
4 B	124 ns
32 B	232 ns
150 B	711 ns
248 B	1,096 ns
Deduced function	$\boldsymbol{L(S) = 4.00 \times S + 107}$ where $L(S)$ is the latency for an S-byte payload in nanoseconds

TABLE V. END-TO-END LATENCIES FOR DIFFERENT PAYLOAD SIZES

Packet size (bytes)	On PTII	On SANDRA
32 B	2,222 ns	2,099 ns
100 B	4,221 ns	4,113 ns
150 B	5,512 ns	5,436 ns
248 B	8,248 ns	8,226 ns
$\overline{\boldsymbol{\Delta}}$	**2.5%**	

the fact that the latency is very low even with this huge relative increase.

B. Second Step: Tuning Latencies

Although the latencies configured within Ptolemy II have led us to a sufficiently accurate end-to-end latency, they are tuned using values corresponding to a fixed-size payload. Although this first step was mandatory in order to check that the overall model was consistent with the SANDRA platform – and it also led us to a major fix for the latter –, we aim to propose a model that is able to compute accurate latencies, irrespective of the payload size. Latencies in some sequential sections are constant, while in others they increase in linear fashion, according to the payload size. For each *sequential section*, we thus measure the corresponding latencies for payloads of multiple sizes. We then deduce the affine functions to compute latencies in the simulator with respect to the payload sizes. Since there are some inaccuracies in measurements, values do not increase in a perfectly linear way. As a consequence, we use simple linear regression to find the best approximated functions. An example is given in Table IV for the `Emission Buffer` sequential section.

Once we have tuned each sequential section in this way, we assess this computation model by cross-checking end-to-end latencies with the SANDRA platform. We use the same scenario as above (one sending of a packet), but with different payload sizes. The ensuing results are given in Table V.

This validates the reliability of our method for building an adaptive though very simple latency computation model.

C. Third Step: Stressed Node

Modelling latencies through several sections will allow us to maintain accurate latencies irrespective of the complexity of the scenario, because, as stated above, each sequential section corresponds to a defined part of the network behaviour.

TABLE VI. END-TO-END LATENCY FOR THE LAST OF 6×240-BYTE PAYLOADS SENT SIMULTANEOUSLY

	Latency (ns)
Calculated	20,311 ns
Measured via Ptolemy II	20,311 ns
Measured via the VHDL simulator	19,332 ns
$\boldsymbol{\Delta}$	**5%**

In a given scenario, each global latency thus results from a combination of sequential sections, taking into account the payload sizes and intermediary queues.

We take as an example the sending of six 240-byte payloads at the same time. We assume, given the payloads size and our previous measurements, that the bottleneck is located at the `AXIread` sequential section. The end-to-end latency for the last payload should then be:

$$L(N) = L_1 + CSS(S) \times (N-1)$$
$$L(6) = L_1 + AXIr(240) \times 5$$
$$L(6) = 20,311\, ns$$

where:

$L(N)$ is the end-to-end latency for the Nth payload,

L_1 is the end-to-end latency for the first payload,

$CSS(S)$ is the latency corresponding to the Critical Sequential Section, i.e. the sequential section where the bottleneck is located, given the payload sizes,

$AXIr(S)$ is the latency corresponding to the `AXIread` sequential section, for an S-byte payload.

We measure this latency on our simulator and the SANDRA VHDL simulator. We have to use this cycle-accurate simulator because we are not yet able to run this scenario on the platform. We obtain the values given in Table VI. The results show the benefit of our architecture, which is still able to compute an accurate end-to-end latency when the scenario is more complex. Of course, this experiment is not sufficient to strictly validate the model irrespective of the scenario, but it shows the flexibility and scalability of our approach.

V. DISCUSSION AND ONGOING WORK

Although we have built several scenarios and cross-checked the simulation model with the SANDRA implementation, the current state of the latter prevents us from assessing the error management feature, i.e. the sending of Packet-Not-Accepted Control Symbols in case of a corrupted packet. We thus plan to develop an error injection feature and build more complex scenarios, e.g. communication via several intermediary nodes or a more stressed network. The presented work helped us to assess our overall model and the benefit of our symbiotic approach.

As we explained in the introduction, it is important that the application clock is consistent with that of the simulator. We

are currently developing a lightweight solution based on open-source virtualization components. This work is not completely finished, but has already shown promising results.

As for the SANDRA platform, we will implement RapidIO DMA logical layer support and integrate it into the DDS ERICA module (see III.D). This DMA logical layer fits our needs better than the messaging one we are currently using.

In the longer term, we aim to design an overall software that makes it easier to conduct experiments with ERICA, for instance by automatically deploying the applications on the SANDRA platform or by generating a Ptolemy II model from a higher level topology drawing. This would also allow us to build virtual networks and machines to logically separate applications that are mapped with different RapidIO nodes, and achieve time consistency.

With regard to the dynamic mesh networking project, research will now focus on real-time QoS cross-layer support, i.e. the design of low-level switching and dynamic scheduling mechanisms and their linking to DDS QoS policies. This work will be done thanks to ERICA.

VI. Conclusion

We are addressing embedded real-time systems whose communications between components tend to become more and more demanding and complex. This reflects the increasing weight of embedded applications in terms of computation and communication requirements. We consider mesh networking of sensors and computing units to be an elegant solution to reconcile real-time constraints with needs in terms of throughput and dynamicity. However, we need a way of enforcing guaranteed real-time QoS from the highest layers down to the mesh network configuration and behaviour. In this paper, we have presented our symbiotic approach to designing this QoS support via a discrete-event Ptolemy II simulated network that is connected with a real implementation of DDS. We have also described how an in-silicon Serial RapidIO platform can help us to define a more representative network model, while the latter may highlight bugs within the implementation. This symbiotic approach enhances our experimentation process and makes it more effective by bridging the simulated and physical worlds. Finally, we discussed the limitations of our approach and presented some ongoing work.

References

[1] RapidIO Trade Association, "RapidIO Specification 3.1," Oct. 2014. [Online]. Available: http://www.rapidio.org

[2] S. Fuller, *RapidIO: The Embedded System Interconnect*. John y & Sons, Jan. 2005.

[3] A. Benveniste, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple viewpoint contract-based specification and design," in *6th Int. Symp. Formal Methods for Components and Objects*, 2008, pp. 200–225.

[4] "FRESCOR: Framework for Real-time Embedded Systems based on COntRacts," Tech. Rep. [Online]. Available: http://www.frescor.org

[5] Object Management Group, "Data Distribution Service (DDS), Version 1.4," Apr. 2014. [Online]. Available: http://www.omg.org/spec/DDS/

[6] [Online]. Available: http://prismtech.com/vortex/vortex-opensplice

[7] [Online]. Available: http://ptolemy.eecs.berkeley.edu/ptolemyII/

[8] R. M. Fujimoto, "Parallel and distributed simulation systems," Dec. 2001.

[9] "IEEE Standard for Modeling and Simulation, High Level Architecture (HLA) – Framework and Rules," *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, pp. 1–38, Aug. 2010.

[10] K. Perumalla, "Parallel and Distributed Simulation: Traditional Techniques and Recent Advances," in *Simulation Conference, 2006. WSC 06. Proceedings of the Winter*, Dec. 2006, pp. 84–95.

[11] S. Strassburger, T. Schulze, and R. Fujimoto, "Future trends in distributed simulation and distributed virtual environments: Results of a peer study," in *Winter Simulation Conference*, Dec. 2008, pp. 777–785.

[12] J. Dahmann and K. Morse, "High Level Architecture for simulation: an update," in *2nd International Workshop on Distributed Interactive Simulation and Real-Time Applications, 1998. Proceedings*, Jul. 1998, pp. 32–40.

[13] C. D. Carothers, R. M. Fujimoto, R. M. Weatherly, and A. L. Wilson, "Design and Implementation of HLA Time Management in the RTI Version F.0," in *Proceedings of the 29th Conference on Winter Simulation*, ser. WSC. IEEE Computer Society, 1997, pp. 373–380. [Online]. Available: http://dx.doi.org/10.1145/268437.268511

[14] G. Lasnier, J. Cardoso, P. Siron, C. Pagetti, and P. Derler, "Distributed Simulation of Heterogeneous and Real-Time Systems," in *2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, Oct. 2013, pp. 55–62.

[15] S. Taylor, "HLA-CSPIF The High Level Architecture COTS Simulation Package Interoperability Forum," in *Proceedings of the 2003 Fall Simulation Interoperability Workshop (WCS)*, 2003.

[16] S. Taylor, X. Wang, S. Turner, and M. Low, "Integrating heterogeneous distributed COTS discrete-event simulation packages: an emerging standards-based approach," *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, vol. 36, no. 1, pp. 109–122, Jan. 2006.

[17] B. Camus, C. Bourjot, and V. Chevrier, "Combining DEVS with Multi-agent Concepts to Design and Simulate Multi-models of Complex Systems," Jan. 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01103892

[18] B. P. Zeigler, T. G. Kim, and H. Praehofer, *Theory of Modeling and Simulation*, 2nd ed. Orlando, FL, USA: Academic Press, Inc., 2000.

[19] J. Vaubourg, Y. Presse, B. Camus, C. Bourjot, L. Ciarletta, V. Chevrier, J.-P. Tavella, and H. Morais, "Multi-agent Multi-Model Simulation of Smart Grids in the MS4SG Project," in *PAAMS'15*, ser. Lecture Notes in Computer Science, vol. 9086, Jun. 2015, p. 12. [Online]. Available: https://hal.inria.fr/hal-01171428

[20] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with cooja," in *Proceedings of the 31th Conference on Local Computer Networks*, ser. LCN '06. Tampa, FL, USA: IEEE Computer Society, 2006, pp. 641–648.

[21] H. Aydt, S. Turner, W. Cai, and M. Low, "Research issues in symbiotic simulation," in *Proceedings of the 2009 Winter Simulation Conference (WSC)*, Dec. 2009, pp. 1213–1222.

[22] [Online]. Available: http://www.opencores.com/project,rio

[23] Bombardier, "Bombardier." [Online]. Available: http://www.bombardier.com/

[24] Xilinx, "Xilinx Zynq." [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[25] ——, "Xilinx." [Online]. Available: http://www.xilinx.com/legal.htm

[26] [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[27] ARM, "ARM Trademark." [Online]. Available: http://www.arm.com/about/trademarks/

[28] ——, "ARM AMBA Trademark." [Online]. Available: http://www.arm.com/about/trademarks/arm-trademark-list/AMBA-trademark.php

[29] J. Kurose and H. Mouftah, "Computer-aided modeling, analysis, and design of communication networks," *IEEE Journal on Selected Areas in Communications*, vol. 6, no. 1, pp. 130–145, Jan. 1988.

[30] H. Mouftah and R. Sturgeon, "Distributed discrete event simulation for communication networks," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 9, pp. 1723–1734, 1990.