Software/Network Co-Simulation of Heterogeneous Industrial Networks Architectures

F.Fummi* S. Martini# M. Monguzzi‡ G. Perbellini# M. Poncino*

* Università di Verona, Verona, Italy
‡ Sitek S.p.A., S.G. Lupatoto, Italy
# Embedded Systems Design Center, Verona, Italy

Abstract

The work presents a modeling and analysis framework for heterogeneous industrial networks architectures which is based on a tight integration of a network simulator with embedded software, middleware and a real-time operating system. This framework is suitable for modeling and simulating the behavior of typical components involved in factory automation applications (e.g., PLCs, remote controllers, operational screens, etc.) when they are connected through heterogeneous industrial networks. Experiments show that the framework allows to take early architectural decisions by evaluating the expected system performance based on the available models.

1. Introduction

Control applications for industrial automation typically follow a hierarchical structure, where slave devices are organized around a master controller. Actuators and sensors, for instance, are usually driven by control software running on the master device.

The typical organization of such heterogeneous networks is depicted in Figure 1, which shows the most important entities involved in a generic factory automation scenario.

In order to simplify the integration, designers leverage existing connectivity support to integrate modules under different configurations. One example of such support is Ultimodule [2], which provides application-proven, pre-assembled components for many factory automation applications that use UltiWIRE, an ad hoc scalable serial link for the interconnection of up to 127 modules. In these environments, a generic backbone interconnect (e.g., UltiWIRE, CAN) interfaces the several subnetworks with different characteristics (speed, bandwidth), communication styles (bus-based, as in Subnet 2 or Subnet 3, or serial, as in Subnet 1), possibly including the connection to a standard LAN. All interconnections between the backbone and the various subnetworks require proper bridges to translate signals between different domains. Bus-based subnets are typically networks of

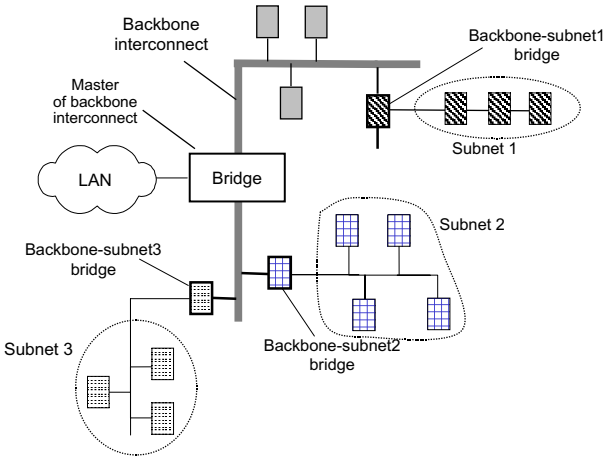

Figure 1. Typical Heterogeneous Network in a Factory Automation Environment.

CANOpen or Profibus ([6]) devices, such as COTS digital I/O modules, that controls time-constrained actuators typical of a plant. Serial subnets may be UltiWIRE or similar networks.

The main master typically runs soft *PLC* (Programmable Logic Controller) applications, with suitable drivers for accessing all the connected networks. A PLC generates cyclic bursts of timed read/write operations on the nodes, that have to be executed within a given deadline (*maximum cycle timing*). Nodes may randomly generate asynchronous events that force the master to invoke services routines.

The above discussion motivates the efforts spent in the community to provide flexible, yet effective, solutions for the simulation of heterogeneous networked environments. Two are the main capabilities required for such solutions:

1. The simulation of the interaction of complex network devices, some of which do execute specific applications which simulate actual devices (such as PLCs or interrupt generators).
2. The timing-accurate simulation of the overall system, so as to be able to meet the real-time constraints.

While the simulation of the sole network infrastructure with the relative protocol stacks can be carried using standard network simulators (e.g., NS-2[5]), the integration of typical applications from the factory automation domain into such frameworks does not have a widely accepted solution. Moreover, timing accuracy in network simulation has been mostly approached from the stochastic point of view, and for TCP-IP based networks; real-time requirements cannot be modeled by conventional network simulators.

In this work we propose a two-step solution to these two challenges, by providing a complete **software-network co-simulation** solution. We first address the first issue, by devising a general solution to the problem of integrating applications into a network simulation environment. We then target the second issue by leveraging the ideas of [3], and incorporate timing accuracy in the co-simulation by exploiting time information available by the actual hardware devices (i.e., the boards).

The proposed environment features both multi-point and field-bus links, and has been applied to real-life Ultimodule-based architectures, including UltiWIRE-driven boards and CANopen field-bus. The experiments show that it can be successfully used for rapid prototyping and exploration of different configurations.

2. Previous Work

Several research efforts have dealt with the problem of modeling and simulating fieldbuses and factory automation systems. The approaches proposed in the literature usually rely on well-established formalisms for representing concurrent systems, possibly including the time dimension. Petri nets (PN) and their variants (timed, stochastic, fuzzy PN) are the most popular solution [7, 8, 9, 10], and are sometimes combined or complemented by the use of concurrent languages such as Estelle or LOTOS [11, 12]. The use of timed PN, in particular, allows to take timing constraints into account.

Other solutions rely on a control-system oriented modeling style, where the system is modeled at a higher level [13], while some others use analytical models for some of the target quantities of the analysis (e.g., available bandwidth or performance) [14, 15].

Even the most accurate of these approaches suffer from one main limitation: the applications that can be modeled using the chosen formalism belong to a limited set of predefined programs, such as sampled systems (i.e., control loops – [14]), or in the best case, PLC applications [8, 10].

None of these schemes is able to model and simulate complex applications which may be implemented by a processor-based board, possibly executing on top of a real-time operating system (RTOS). This limitation is precisely the target of this work, in which we propose an extension of an existing co-simulation methodology [3] that allows to integrate a model of software with a model of a network.

3. Modeling

The network scenario shown in Figure 1, when applied to the the factory automation target of this work, results in the protocol architecture of Figure 2.

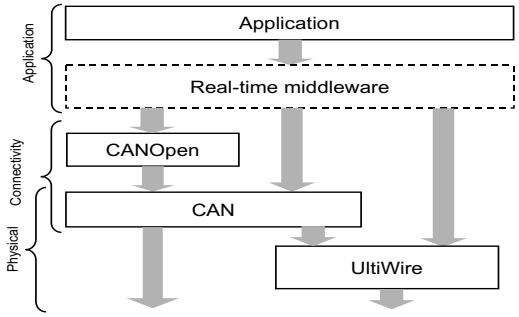

Figure 2. Protocol Architecture of the Target Network.

We define three rough layers: *application*, *connectivity* (i.e., data-link/network), and *physical*. The picture clearly shows the non-layered nature of the involved protocols. As a matter of fact, the two bottom levels may partially overlap, depending on what parts of a protocol are included.

The physical level protocol can either be UltiWIRE or the physical layer of CAN, although CAN (its upper levels) may use UltiWIRE as a backbone interconnection. For the connectivity layer, the only strict requirement is that CANOpen must rely on CAN for its services. The application layer may rely either on a full CANOpen/CAN stack or directly on the physical layer protocol, and may or may not include a real-time middleware (the dashed box) layer.

Modeling this heterogeneous scenario requires a full-fledged network simulation environment, which, however, is by definition suitable to model only the two bottom layers. Incorporating the application layer in the simulation needs a more general solution.

The contribution of this work is two-fold. First, we provide a general co-simulation solution based on (i) extending an existing network simulator (in this work, NS-2 [5]) so as to model all the various protocols (UltiWIRE, CAN, and CANOpen) involved in our target environment, and (ii) finding a suitable way to connect the applications to the network simulation environment. This solution has the advantage of offering an homogeneous way to simulate the entire system, but cannot be used to model time-related properties of the application (essential in factory automation), such as the real-time middleware of Figure 2.

The second contribution is the solution of this limitation. In particular, we discuss the role and the use of an instruction-set simulator (ISS) and a board to improve the overall modeling methodology thus allowing to interface an application (executed by an ISS) and a generic simulated network (i.e., the NS network), where the application layer is replaced by the actual application.

4. All NS Modeling

The first solution for the simulation of the architecture of Figure 1 consists of using NS-2 [5] to build the network topology and to model and simulate the various devices. NS-2 is a discrete event simulator that provides support for the simulation of an IP-based protocol stack, in which a number of network and data-link protocols are available (IP, TCP, UDP, Ethernet, etc.).

NS-2 provides easy extension to the users, by separating the network configuration phase (described through a Tcl script) and the actual simulation engine, implemented in C++.

The main entities involved in a simulation are *nodes*, *links*, *agents* and *applications*; each one is a model of a real object. For instance, a node in NS-2 can be used to represent a node in a real network. Every object has a set of attributes, (e.g., a link connects two nodes and it has a bandwidth and a delay). To add a new protocol in NS-2, the programmer must write a new class derived from the four above objects. For instance, to add a new transport protocol it is necessary to write a new `Agent` object. Once the new class has been written, it becomes part of the compiled hierarchy. Figure 3 shows the class hierarchy that needs to built to model the overall network architecture of Figure 1.

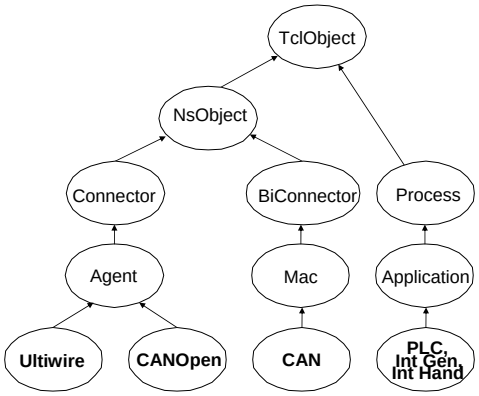

Figure 3. Extended NS-2 Class Scheme.

4.1. UltiWIRE

An UltiWIRE network is a daisy chain network with one master and one or more slaves. The master is responsible for initiating all communications over the network; slaves can only communicate with the master, but not among them. The network requires one single-ended signal for the interconnection between network nodes (master or slaves) over short distances, while in the case of long distances a different signal is required.

Communication evolves around the exchange of two basic frame types: Tx and Rx. Frames are 16-bit words and basically contain a command, data (for writes) and CRC, plus some other lower-level info. A typical communication cycle starts with the master sending a Tx frame to all slaves. The selected slave executes the TX frame command, and it replies by sending an RX frame to the master. Special arrangements for error handling as well as broadcasting are supported. Further details can be found in [4].

The steps required to model UltiWIRE under NS-2 are (i) defining two types of frame types (Tx and Rx), and (ii) modeling the protocol by creating two new `Agent`s, one for the master (`UltiWIREmaster`) and one for the slave (`UltiWIREslave`). Each UltiWIRE node is modeled with an `Agent` attached on an NS-2 node: the agent can be the `UltiWIREmaster` or the `UltiWIREslave`.

4.2. CAN and CANopen

CAN is a serial bus system suitable for the interconnection of smart devices. CAN consists of three layers: *object* layer, *transfer* layer and the *physical* layer. The object and the transfer layer include all services and functions provided by ISO/OSI data-link layer, while the physical layer is in charge of transferring the bits between the different nodes with respect to the all electrical properties.

Therefore, the NS-2 model of CAN protocol is a C++ class extending the Media Access Control C++ class of the NS-2 hierarchy. The NS-2 CAN object implements CSMA/CD to handle access to the shared bus. CANopen is a CAN-based higher layer (ISO/OSI layer 7) protocol originally developed for industrial control systems. CANopen is built around the central concept of an *Object Dictionary* (OD), the interface between the application and communication, defined within each device, to define every function, variable and data type seen via the network.

The NS-2 model of the CANOpen protocol is a C++ class derived from the NS-2 `Agent` class. To simulate a CANopen Network a new NS-2 Agent called `CANopenAgent` have been defined. This new agent models the nodes on a CANopen network. The `CANopenAgent` provides the features to support a desired traffic profile. This agent has a local memory to store data. The `CANopenAgent` implements two operations to read/write from/to the memory of another CANopen node. These operations are exported to the applications running on it.

4.3. PLC and Interrupt Generator

A typical industrial application is a PLC (*Programmable Logic Controller)*; the typical pattern of a PLC is to generate cyclic burst of timed read/write operations to the networked nodes, that have to be executed within a given time threshold.

The NS-2 model of the PLC is a C++ class derived from the NS-2 `Application` class. An application is attached on an agent. It can generate and send/receive data through the underlying agent, or it can invoke the agent's methods to access and execute the services of the agent's protocol.

Any PLC application has a local memory to store data. The NS-2 model reads a *program* from a file, which typically

consists of a sequence of `read`, `wait` (a specified amount of time) and `write` operations that are iterated in a cyclic fashion. Finally, the NS-2 PLC application calls the corresponding operations on the underlying agent, e.g., the `UltiWIREmaster` agent. The PLC Application object is parameterized with respect to (i) the PLC memory size, and (ii) the operation burst.

Another application used to increase the traffic on the network is an *Interrupt Generator (IG)*. This application models the interrupt behavior of an UltiWIRE device. The interrupts are used by a UltiWIRE slave to inform the master of an event. The IG embeds a parametric random number generator to generate interrupts at random times during the simulation. The user can configure the properties of the application, such as the type of the random distribution. The IG interrupts the master by setting a proper bit in the Rx frame; when the Rx frame reaches the master, the interrupt polling routine is invoked to determine which slave has generated the interrupt. The master calls then the ISR (Interrupt Service Routine) associated to the slave that has generated the interrupt. This ISR is modeled by a *Interrupt Handler* application attached to the UltiWIRE master agent.

The user has to define a PLC-like program (a sequence of read, write and wait operations) to be loaded by the *Interrupt Handler* Application, and associate this program to a slave. This program will be executed whenever that slave generates an interrupt.

5. Board-Based Modeling

Once the network has been modeled using the described framework, the designer might be interested in applying realistic workloads by plugging real industrial applications into the network. For instance, one may replace the NS-2 implementation of a PLC with a real *soft PLC* application, i.e., a program that runs on an ordinary computer and mimics the operation of a standard PLC. In this case, we have to establish a communication between the actual hardware controller running the PLC application and the NS-2 simulator, modeling the rest of the network topology. Morever, most real-life applications execute using the functionalities offered by an operating system, which provides a set of APIs through which the devices can be accessed.

If, in our simulated network scenario, we want to replace a simulated application with a softPLC, we have to modify the simulator's kernel as well as the driver of the actual device used by the application. The new driver would not use the real hardware device, but rather it has to communicate with the simulated device (i.e., the simulator). This is the solution proposed in [3], which we leverage to establish the communication between NS-2 and a generic application. In this work, we use the methodology of [3], so that the network is modeled as a device, accessible through a suitable device driver of the host operating system. Another advantage of this scheme is the seamless replacement of the ISS with a board running the real-time application.

5.1. Programming Model

In order to establish a communication between the application and NS-2, the programmer must implement a driver for the host operating system, which controls the NS-2 simulation. The driver consists of (i) the code that handles the interaction with the external device through proper channels, (ii) the ISR to handle interrupts, and (iii) a suitable API that allows to interact with the driver from the application source code.

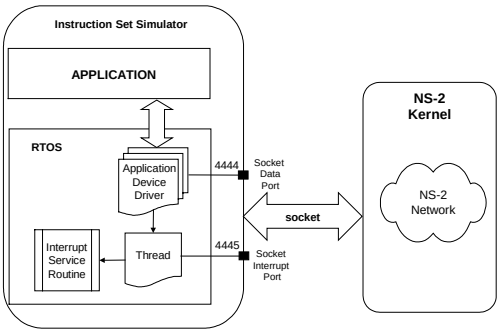

Figure 4. Programming Model Architecture.

In our case, the driver operates as follows (Figure 4):

1. It communicates with the NS-2 kernel through port 4444 (called the *socket data port*) by sending a proper message based on Application protocol;
2. It creates a thread that listens to the interrupts generated from the NS-2 simulation; interrupts are received through port 4445 (called the *socket interrupt port*). When an interrupt occurs, the ISR written by the programmer, has to be started to manage the interrupt.

The programmer uses the methods of the driver API to build a communication between the application and the NS-2. This communication is made possible by properly modifying the NS-2 scheduling algorithm. The driver and the NS-2 kernel communicate by exchanging messages on a communication channel and on an interrupt channel for the interrupt handling.

Figure 5 shows the extra checks done by the NS-2 scheduler for implementing the driver-kernel co-simulation mechanism. At the beginning of a new simulation cycle, the NS-2 kernel checks the channel; if there is no incoming message, the kernel does the normal handling of the events in the scheduler queue. At the end of the event scheduling, before moving to the next simulation cycle, it verifies if an interrupt has been generated. In this case, the interrupt is notified to the driver by sending a message on the reserved interrupt channel. Whenever a message arrives, the kernel reads this message and processes it in according to the application protocol.

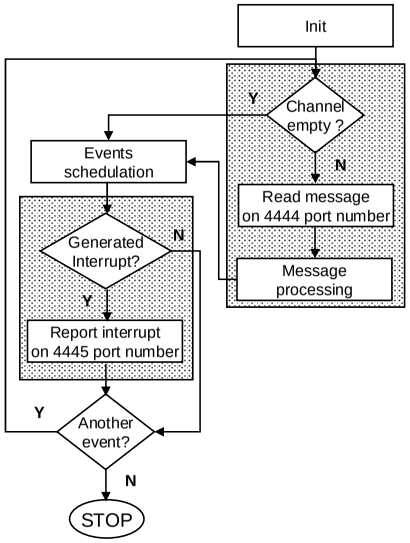

Figure 5. Modified NS-2 Scheduling Algorithm.

6. Performance Analysis

The two proposed co-simulation strategies (the *all-NS* and the *board-based* ones) provide different tradeoffs between simulation speed and simulation accuracy, which will be discussed in this section.

Table 1 shows the performance of the two proposed simulation schemes. The three columns refer to different amounts of simulated times (1000, 10000, and 100000 seconds). The table clearly shows how the *All-NS* scheme outperform (roughly 50% faster) the *board-based* one.

	Simulated Times [ms]		
	1000	10000	100000
All NS Modeling	2090	21420	227120
Board-Based Modeling	4877	40175	359428

Table 1. Simulation Performance Results.

The latter scheme, however, allows to replace the application layer with a more complex application, with real-time constraints and even including an operating systems.

We tested the overall methodology on a typical configuration of an industrial network, consisting of a CAN/CANOpen bus performing hardware PLC (Figure 6). In order to implement this environment using a proprietary protocol, namely, UltiWIRE, we have to replace the HW PLC with a software solution, if no hardware PLC exists for the proprietary protocol.

The software solutions are the following:

1. Use a standard soft PLC, such as ISaGRAF [16];
2. Use an ad-hoc C/C++ application that emulates the behavior of the hardware PLC.

These two solutions must guarantee the same behavior as the hardware PLC, in terms of time constraints, measured using the *maximum cycle time* associated to a PLC model.

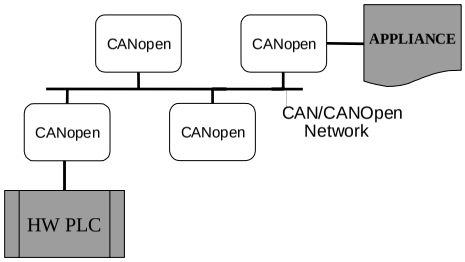

Figure 6. Example Industrial Network.

6.1. All-NS Modeling

The scenario shown in Figure 6 is first modeled using NS-2 components, as described in Section 4. Figure 7 shows that the evolution over time of the execution of the PLC application. We notice that all PLC cycles are within the maximum cycle time, and we can thus say that the network is able to support the workload generated by the PLC and the same behavior must be implemented by the SW solutions.

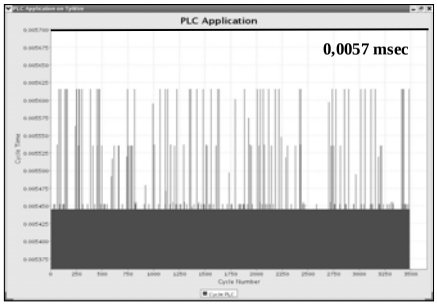

Figure 7. Cycle Time Profile for the All-NS Model.

6.2. Board-Based modeling

To improve the accuracy of the simulation we model the scenario of Figure 6 using the methodology presented in section 5 in which a real application is used instead of the simulated one. We will examine two possible solution: the use of a soft PLC software and the use of a standard C/C++ application.

6.2.1. Using a Soft PLC In the first case study the hardware PLC is replaced by ISaGRAF, as shown in Figure 8. ISaGRAF is a widely used, IEC 61131-3 compliant control software environment for creating distributed control systems. It is essentially an interpreter of a user program (the ISaGRAF program) written by using a ISaGRAF development environment and debugger tool.

A typical workflow of ISaGRAF (and usually of any other soft PLC) is a cyclic execution of these operations: (i) scanning the physical inputs of the process to drive; (ii) processing application data according to the ISaGRAF application programs; (iii) performing physical outputs update.

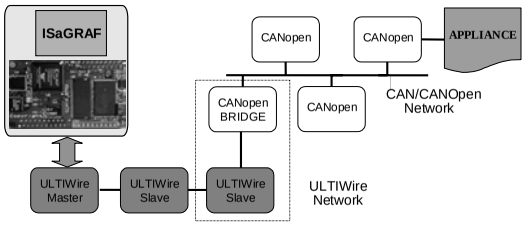

Figure 8. ISaGRAF as a Substitute of a Hardware PLC.

The PLC application running on the TpiCU™ SCM20 board [17] is attached to the UltiWIRE NS-2 model via a device driver, using the co-simulation methodology presented in Section 5. This PLC application reads/writes data from/to the *Appliance* on the CANopen node. The CANopen network and the UltiWIRE bus are connected together through a UltiWIRE-CAN bridge, which receives UltiWIRE frames from the UltiWIRE master and executes the requests on a CANopen node.

This approach allows to solve the problem described at the beginning of this section with a minimal effort, due to use of a user-friendly solution provided by the ISaGRAF development environment. On the other hand, this approach is expensive from a CPU load point of view, since the ISaGRAF solution consists of an interpreted code. Because of this overhead, time constraints cannot be satisfied, as shown in Figure 9.

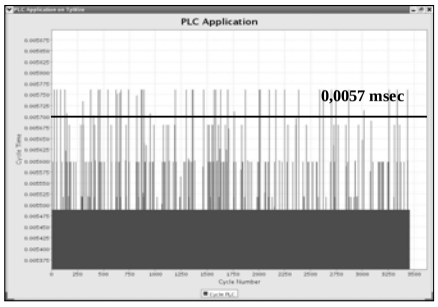

Figure 9. Cycle-Time Profile for the ISaGRAF PLC Solution.

6.2.2. Using a Generic Application There are two main solutions to the previous problem: (i) enlarge the bandwidth of the UltiWIRE channel or (ii) reduce the load due to the software. We focus on the second solution (Figure 10).

Simplfying the software overhead allows to satisfy the PLC cycles constraints, and the cycle-time profile becomes again close to that of Figure 7. Notice, however, that writing an ad-hoc C/C++ code modeling a PLC application could be a non-trivial task. Therefore this solution trades efficiency for programming effort.

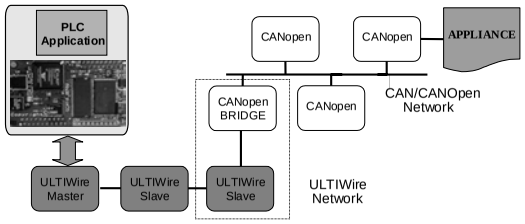

Figure 10. C Application as a Substitute of a Hardware PLC.

7. Conclusions

Analyzing heterogeneous networks such as those commonly found in industrial automation is a challenging task. The use of network simulator is not a solution, however, because it does not allow to model timing constraints and to integrate real-life applications. In this work, we have presented a heterogeneous modeling solution that allows to integrate models of a network with models of software and hardware, and to effectively co-simulate them.

References

[1] CAN In Automation, `http://www.can-cia.de`.

[2] Ultimodule Inc., `http://www.ultimodule.com`.

[3] F. Fummi, S. Martini, G. Perbellini, M. Poncino, "Native ISS-SystemC Integration for the Co-Simulation of Multi-Processor SoC", *DATE'04*, Feb. 2004, pp. 564–569.

[4] N. Drago, et al., "Estimation of Bus Performance for a Tuplespace in an Embedded Architecture". *DATE'03*, Mar. 2003, pp 188–193.

[5] L. Breslau em et al. "Advances in Network Simulation", *IEEE Computer*, Vol. X, No. 5, May 2000, pp. 59–67.

[6] Profibus In Automation, `http://www.profibus.com`.

[7] S. H. Hong, S. G. Lee, "Performance Analysis of the Data Link Layer in the IEC/ISA Fieldbus by Simulation Model," *ETFA'96*: Nov. 1996, pp. 593–601.

[8] G. Marschall, "Petri Net Simulation of a Fieldbus Communication Application," *ETFA'96* Nov. 1996, pp. 63–69.

[9] G. Noubir, P. Raja, J.D. Decotignie, "Simulating the Fieldbus Synchronous Model by Timed Petri Nets," *IECON'94:* Sep. 1994, pp. 1205–1210.

[10] A. Di Stefano, D. Mirabella, "Evaluating the Field Bus Data Link Layer by a Petri Net-Based Simulation," *IEEE Transactions on Industrial Electronics*, Vol. 38, No. 4, Aug. 1991, pp. 288–297.

[11] L. Hohwiller, S. Wendling, "Fieldbus Network Simulation Using a Time Extended Estelle Formalism," *MASCOTS'00:* Aug/Sep. 2000, pp. 92–97.

[12] L. Durante, R. Sisto, A. Valenzano, "Integration of Time Petri Nets and TE-LOTOS in the Design and Evaluation of Factory Communication Systems," *IEEE International Workshop on Factory Communication Systems*, Oct. 1997, pp. 71–80.

[13] H.F. Abdel-Ghaffar, M.F. Abdel-Magied, M. Fikri, M.I Kamel, "Performance Analysis of Fieldbus in Process Control Systems," *American Control Conference*, Jun. 2003, pp. 591–596.

[14] P. Raja et al., "Synchronous Model for Fieldbus Applications," *IECON'93:* Nov. 1993, pp. 525–529.

[15] M.D. Rubio Benito, J.M. Fuertes, E. Kahoraho, N. Perez Arzoz, "Performance Evaluation of Four Field Buses," *ETFA'99*: Oct. 1999, pp. 881–890.

[16] ICS Triplex ISaGRAF, `www.altersys.com`.

[17] TpiCU, Theseus Project Integrated Control Unit, `http://www.embedding.net/icu`