# Including Real-Life Application Code into Power Aware Network Simulation


Georg Möstl
moestl@riic.at

Richard Hagelauer
hagel@riic.at

Research Institute for Integrated Circuits
Johannes Kepler University
Linz, Austria

Andreas Springer
a.springer@icie.jku.at

Gerhard Müller
g.mueller@icie.jku.at

Institute for Communications and RF-Systems
Johannes Kepler University
Linz, Austria



## ABSTRACT

We present a methodology and a toolset for power aware HW/SW co-simulation including real-life application code at network level.

The toolset consists of the known OMNeT++ network simulation environment and the PAWiS framework, which was extended to include time-annotated and natively executing C code, and allows detailed analysis of the power consumption of single modules in the network. In conjunction with the support of interrupt handling, this especially addresses the needs of applications running on nodes of wireless sensor networks (WSNs). The presented partitioning of the application into platform-dependent and platform-independent SW layers provides easy porting of the simulated code to real sensor nodes. Therefore the established simulation environment supports the development, implementation and verification of energy optimized protocols for real-time industrial applications using WSNs.

To demonstrate the functionality of this approach, the methodology was applied to a simple real-world networking test scenario and the achieved simulation results are compared to real-world measurements.



## Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*; I.6.3 [**Simulation and Modeling**]: Applications; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Discrete event*

## General Terms

Design, Verification



Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.



*SIMUTools 2010* March 15–19, Torremolinos, Malaga, Spain.



Copyright 2010 ICST, ISBN 78-963-9799-87-5.



## Keywords

Co-simulation, network level, power aware simulation, real-life code execution


## 1. INTRODUCTION

Design, evaluation and testing of wireless sensor networks (WSNs) are often accomplished with the help of network simulators. Among several different simulators ([8], [13]) there is OMNeT++ [11], a C++ based discrete event simulation environment specifically used for network simulation.

Although OMNeT++ is a powerful tool, the simulation of WSNs must consider special aspects which are not directly addressed in this environment. WSNs are mainly used for monitoring and controlling automated processes in various application areas including industry, agriculture and home entertainment. Nodes of such networks are required to have very low power consumption in order to achieve lifetimes of several years supplied by single batteries. This can only be accomplished with the help of a power aware simulation of the network. It is crucial to get a detailed profile of the power consumption of single modules within the nodes, as this enables the designer to tune and optimize these blocks.

CASTALIA [9] is an extension of OMNeT++ that offers energy aware simulation using a resource manager module. This module keeps track of various node resources, the most important of all being energy. Every module within the node can report its actual energy consumption to the resource manager.

Another framework based on OMNeT++ capable of logging the power consumed by a wireless node is PAWiS [12]. PAWiS not only enables the designer to protocol the power consumption of every module within a node, but also provides several different power reporters (e.g. constant, resistive, linear and user-defined reporters). A complete hierarchical model of power suppliers and consumers (i.e. a simple electrical network) is established during simulation. In addition PAWiS offers a clear separation of HW and SW tasks, a basic CPU model and support for interrupts.

Although PAWiS allows to model a WSN at the network and system level, it lacks the integration of real-life code into the simulation. In the following sections, a methodol-


*Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8643*
*http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8643*

ogy is presented which eliminates this drawback and makes possible a power aware simulation of WSNs, including time-annotated real-life application code. This application code can be ported easily to real sensor nodes.

The remainder of this paper is organised as follows: Section 2 gives a short review of related work done so far. Our concept is introduced in section 3, the usage of the concept and exemplary simulation results are presented in section 4. Section 5 comprises the conclusion of this paper and future work.

# 2. RELATED WORK

Most publications concentrate on the usage of heterogeneous environments that consist of different simulation engines to co-simulate HW and SW components at the network level.

In [5], the NS-2 network simulator is used in conjunction with an instruction set simulator (ISS). As presented, this provides the possibility of simulating heterogeneous networks that contain different wired protocols (UltiWIRE, CAN, CANOpen) in factory automation environments including real industrial applications. Timing-accurate simulations can be run with the drawbacks of (a) modifying the NS-2 scheduling algorithm, (b) writing an application driver module and (c) degrading of simulation performance (roughly 50% slower than network simulation alone).

[6] presents a three-tier scheme consisting of SystemC to model HW at the system level, an instruction set simulator to execute SW and NS-2 to model networks. This scheme and the methodology of HW/SW/network partitioning are applied to the design of a system-on-chip that performs the fast path of IPv4 routing.

In [3], a client/server co-simulation environment is presented. The server, which is controlled by the client through SOAP services, consists of SystemC, $\mu$CSim (ISS for the Intel 8051 microcontroller) and NS-2.

Although all these solutions offer a timing-accurate simulation of networks including real-life applications, they have serious drawbacks. First, the simulation performance of the heterogeneous environments is decreased compared to homogeneous simulation environments. Secondly, the energy consumption of nodes and their building blocks contained in the network is neglected by these approaches. The goal of this work is to establish a power aware, fast and accurate simulation environment for WSNs using only one simulation engine and native execution of time-annotated SW.

A similar concept regarding time-annotated simulation of real-life code is presented in [2]. The SW consisting of the operating system (device drivers, ISRs, etc.) and application code is executed natively on the host machine. HW is described in SystemC and linked to the SW by a timed bus functional model. Therefore, it is possible to run a fast and accurate HW/SW co-simulation at the system level.

# 3. METHODOLOGY

## 3.1 Software architecture

Code portability is a key issue in the development of applications, especially when the targeted platform is a microcontroller. This is due to the huge variety of microcontrollers, which differ not only between vendors but also between the families a vendor offers.

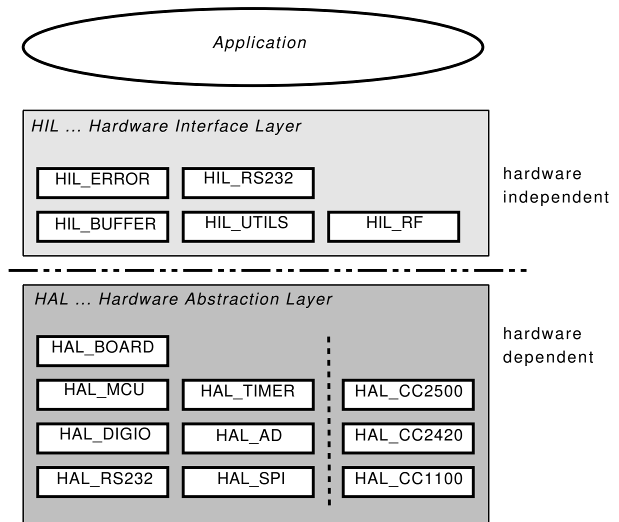


**Figure 1: Software architecture**

In our case, in addition to the different microcontroller architectures, there is another target platform - the simulation. Seamless porting of the application between these platforms can only be achieved by a SW architecture which splits the code into a HW/platform-dependent and HW/platform-independent part. As shown in Figure 1, this is accomplished by using different software layers.

The hardware abstraction layer (HAL) comprises the platform-dependent code for various HW modules. For example, the initialisation of an SPI module varies between different microcontrollers and is therefore encapsulated in the HAL_SPI block, which is part of the HAL. As a result, switching to another platform is as simple as exchanging the HAL_SPI block. Currently, HALs are available for MSP430FG4618, MSP430F2274 and the PIC18x microcontroller families.

The hardware interface layer (HIL) supports hardware independency at a higher abstraction level and therefore provides higher reusability. An integral part of the HIL is an error detection mechanism that gives feedback to the application. In summary, the HAL implements functions which are directly supported by the hardware, whereas the HIL uses them to serve as a module handler to the application.

## 3.2 Creating shared libraries

The structured application code is used to build shared libraries, which are then linked with the shared libraries of the simulator to the simulation executable.

One shared library must be built for each application class running on a node in the network. This gives the advantage of defining different preprocessor constants for different applications, which makes possible powerful conditional compilation. For example, if node 1 uses a CC2500 module and the corresponding API for running an application, but node 2 needs a CC2420 API, all one has to do is to include the switch "-DCC2500" or "-DCC2420" in the compilation scripts. These scripts are generated automatically from a list of preprocessor constants and a list of source files specified in a text file. The HAL files targeting the microcontroller must not be used in simulations, i.e. they are excluded from the build step by not listing them in the

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8643
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8643

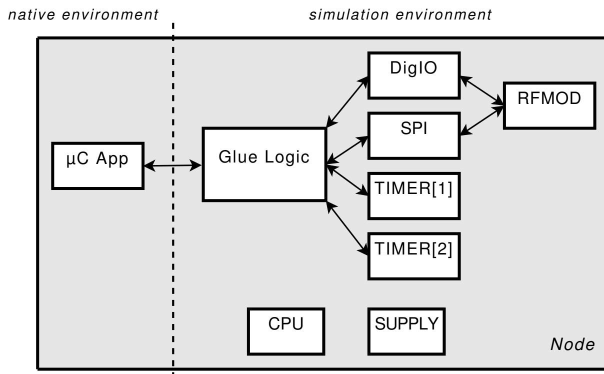


**Figure 2: Node structure**

text files. Instead, HAL source files corresponding to the simulation are specified in the text files.

To avoid name clashes between the several linked application libraries, every library has its own namespace. This namespace is created automatically during the compilation step, using macros and preprocessor constants defined in the scripts.

## 3.3 Annotating the application code

Differentiating between wall clock and simulation time, native execution of application code takes zero simulation time. To run a timing-accurate simulation, synchronisation between natively executed code and network simulation is necessary. This is achieved by the global function *requireCpu()* provided by the *Glue Logic* module. Calling this function suspends the execution of tasks and gives control to the simulation kernel by using PAWiS functionality. After the specified simulation time duration has expired the suspended task is reactivated.

Annotating the application source code with calls to *requireCpu()* gives block-level time estimates. The execution duration of the block according to a norm CPU and a profile of the statements classified into integer, float, memory and control operations must be specified. These formal parameters can be used in *requireCpu()* to scale the execution time with respect to the targeted CPU (i.e. microcontroller).

To determine the execution time of several lines of C code, a macro is provided. It uses “C code to assembler” and “assembler to clock cycles” ratios together with the specified clock period to compute the time parameter for *requireCpu()*. An improvement to this rather simple approach can be achieved by using mathematical models found in [4].

## 3.4 Glue Logic

Figure 2 shows the structure of a WSN node model with the application, which is executed natively, on the left side and the modelled HW modules, which are interpreted by the simulator, on the right side. Between these two worlds resides the *Glue Logic*, which links them together. Although these modules are described at a high abstraction level using C++, they reflect reality regarding the functional, timing and power consuming behaviour.

Communication between *Glue Logic* and HW modules is achieved through PAWiS functional interfaces which build on the OMNeT++ message passing system. A tighter coupling between the application and the *Glue Logic* is achieved

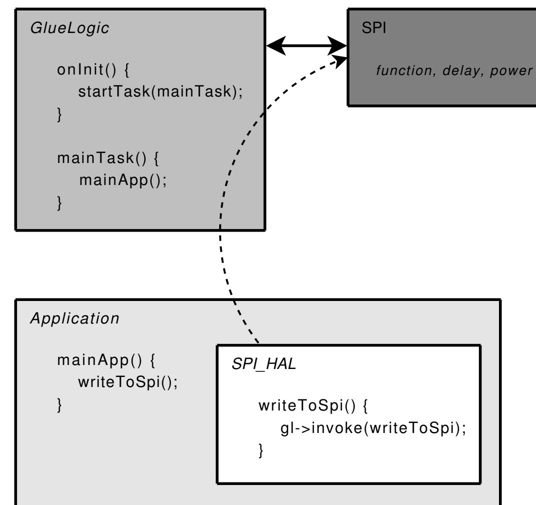


**Figure 3: Calling sequence**

trough function calls (the same holds for connections to the CPU and supply module). The structure and connections of a node are specified through network description (NED) files as supported by OMNeT++.

Four main tasks have to be accomplished by the *Glue Logic*. First, a database of registered interrupts and pins (in/out) is contained in the module. Every HW module registers its symbolic names used for interrupts or pins during the initialisation step of PAWiS to the *Glue Logic*. The mapping of symbolic names to names used by the application is specified in the NED file or in the initialisation file of OMNeT++. This yields a greater flexibility of the simulation setup. For example, a transceiver model (e.g. CC2500) can be written for the symbolic name GDO2 and can be used in conjunction with an application which expects the usage of PORT2.7. Changing either the HW module (e.g. GDO2 → GDO5) or the application (e.g. Port2.7 → Port4.3) leads (in the simulation) only to a change of the initialisation parameters.

Secondly, the *Glue Logic* also provides interrupt handling, as the module serves as a wrapper for interrupt service routines (ISR). Every interrupt is linked to this wrapper in the initialisation phase of the HW modules. After an interrupt occurs, the *isrWrapperTask()* method in the *Glue Logic* is started, and subsequently calls the corresponding ISR of the application code. The interrupt behaviour resembles that of real HW, i.e. application execution is suspended, the ISR is executed and afterwards application execution is resumed. Nested interrupts are also supported.

Thirdly, the *Glue Logic* of every node is responsible for starting the application as depicted in Figure 3 using pseudo code. In the initialisation step, PAWiS calls the *onInit()* method of every module contained in the simulation environment. In this method the *Glue Logic* starts a new task which subsequently calls the top level function of the application. Thus, for every class of application there is a specialized *Glue Logic* (e.g. PawisGlueLogicTestApp) module which subclasses from a base *Glue Logic* (e.g. PawisGlueLogicCApp) class and overrides the *onInit()* and *mainTask()* methods.

Finally, the *Glue Logic* delegates calls from the applica-

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8643
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8643

tion code to the corresponding HW modules through PAWiS functional interfaces (see Figure 3). An example is given in Figure 3 where the application writes data over an SPI connection. The application calls the corresponding API function of the SPI_HAL block. Porting of the SPI_HAL block to the simulation platform forwards this call to the *Glue Logic* module, which invokes the functional interface of the required SPI module.

# 4. RESULTS

To demonstrate the functionality and usefulness of the presented methodology, we applied it to a real-world test scenario. The scenario consisted of a sensor node sending a data packet regularly to a base station, which writes the RSSI (received signal strength indicator) value of the correctly received packet (CRC check is good) to an RS232 interface. In practice, the base station would be connected to a PC, which would process the RS232 bytes further. As this is outside the scope of the simulation the RS232 data is logged to a text file by the RS232 HW module.

Both the sensor and the base station node used a CC2500 transceiver module, which was modelled in rich detail (using fsm, fifos, registers, etc.). As in reality, this model must be accessed through SPI commands and triggers interrupts on the GDOx pins. This ensures easy portability of the application code between simulation and real sensor nodes. Using a toggling LED, each node displays successfully sent or received packets.

The current version of PAWiS (v2.0) used for wireless communication between sensor nodes only provides a free space propagation model that neglects large and small-scale fading. Futhermore, PAWiS v2.0 only supports OMNeT++ version 3.3 or below. The simulation executable was built using the GNU compiler collection (GCC v4.2).

PAWiS logs the power consumption of all modules to a single text file in a proprietary format. Octave (v3.0) scripts are provided which process this file and, using Gnuplot (v4.2.2), plot the power and energy consumption of individual modules. Special filter phrases for module names and a time-range for plotting can be specified. For example, Figure 4 shows the power consumption of the RF module voltage regulator at the base station in the time-range of 1.078 s to 1.081 s in the simulation. After a packet consisting of 2 bytes of preamble, a 4-byte synchronisation word, 2 bytes payload and 2 bytes CRC has been received successfully using 2-FSK modulation at a datarate of 2.4 kBaud, the CC2500 transceiver switches from the receive state (*RX* state) to the *IDLE* state. Thus, power consumption drops to 4.5 mW (at 1.0792 s). To receive further packets, the application triggers an *SRX* strobe on the SPI which causes a state change from *IDLE* (through several intermediate states) to *RX*. Consequently, power consumption rises to approximately 43 mW. Current consumption values were extracted from [7] for a supply voltage of 3.0 V.

The simulated power consumption of Figure 4 can be easily compared to real-world measurments using Figure 5 which depicts the measured current consumption (i.e. the voltage drop measured on a 74.99 $\Omega$ resistor) of the RF module voltage regulator at the base station. Knowing the supply voltage of 3.0 V and the current consumption, the power consumption in different states can be calculated.

Figure 6 visualizes the energy consumption of all consuming modules contained in the base station (i.e. Node1) for

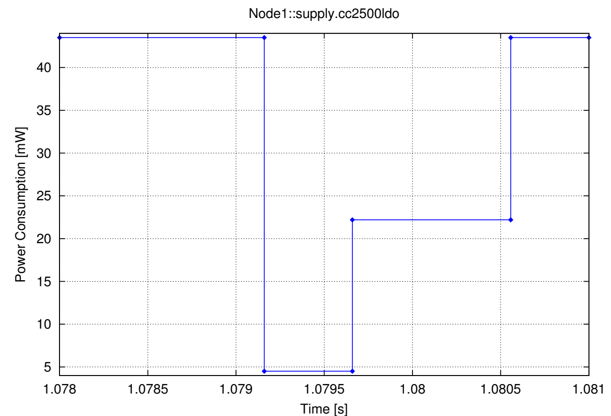


**Figure 4: Simulated power consumption of RF module**

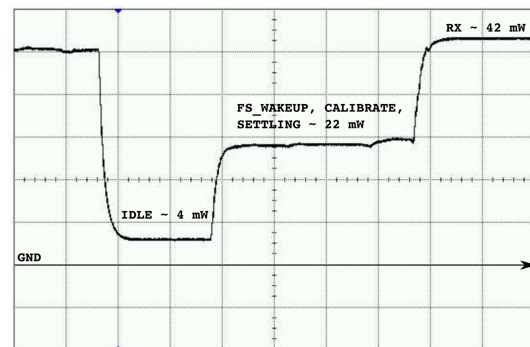


**Figure 5: Measured current consumption of RF module (200 $\mu$s/div; 200 mV/div)**

*Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8643*
*http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8643*

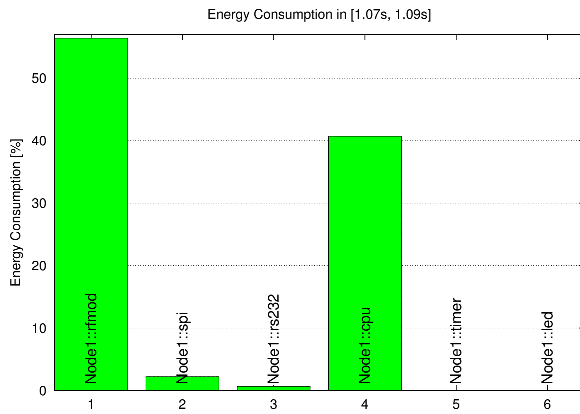


**Figure 6: Energy consumption of modules**

```
Running simulation...
** Event #0  T=0.0000000   ( 0.00s). CC2500Net.Node1.gluelogic (id=12)
PawisGlueLogicTestApp::mainTask() entered!
call rf_test_ban()
rf_test_ban() entered
rf_test_ban() initialized
** Event #1  T=0.0000000   ( 0.00s). CC2500Net.Node2.gluelogic (id=24)
PawisGlueLogicTestApp::mainTask() entered!
call rf_test_acn()
rf_test_acn() entered
** Event #2  T=0.0000000   ( 0.00s). CC2500Net.Node1.cpu (id=9)
** Event #3  T=0.0000000   ( 0.00s). CC2500Net.Node2.cpu (id=21)
** Event #4  T=6.00000000e-06 (  6us). CC2500Net.Node1.cpu (id=9)
** Event #5  T=6.00000000e-06 (  6us). CC2500Net.Node2.cpu (id=21)
```

**Figure 7: Simulation without time annotation**

a specified time interval. This value is then related to the total consumed energy of the node in this time interval. As shown in Figure 6 the RF module of Node1 needs approximately 56% of the energy consumed in the time-range of 1.07 s to 1.09 s.

The effects of simulating time-annotated code are visualized in Figures 7 and 8, which depict snippets of OMNeT++ transcripts. Simple print statements in the application code are used to point out the advantage of time annotation. Without time annotation, the native execution of the *rf_test_ban()* function lasts for zero simulation time and therefore the strings "rf_test_ban() entered" and "rf_test_ban() initialized" are printed at the same point in time. Annotating the initialization block of *rf_test_ban()* inserts a (simulation) time interval between these two print statements (see timestamps 0 $\mu$s and 11 $\mu$s). Consequently, using time annotation yields simulations with higher temporal accuracy.

After the real-world networking scenario was simulated and the RSSI values at different distances were generated, the same application was transferred to a MSP430 microcontroller architecture by only exchanging the HAL. The measurements for the RSSI values were made in a free space environment with both transceivers mounted on 1.2 m high tripods and transmitting at a power of 1 dBm. Characteristics of the employed antenna can be found in [1].

Figure 9 shows the results obtained from calculation (RSSI_c), simulation (RSSI_s) and measurements (RSSI_m). Simulation results are consistent with calculated data, although the resolution in the simulation was limited to 1 dBm. Real-world measurements deviate from calculation using Friis equation for simple free space propagation, which can be ex-

```
Running simulation...
** Event #0  T=0.0000000   ( 0.00s). CC2500Net.Node1.gluelogic (id=12)
PawisGlueLogicTestApp::mainTask() entered!
call rf_test_ban()
rf_test_ban() entered
** Event #1  T=0.0000000   ( 0.00s). CC2500Net.Node2.gluelogic (id=24)
PawisGlueLogicTestApp::mainTask() entered!
call rf_test_acn()
rf_test_acn() entered
** Event #2  T=0.0000000   ( 0.00s). CC2500Net.Node1.cpu (id=9)
** Event #3  T=0.0000000   ( 0.00s). CC2500Net.Node2.cpu (id=21)
** Event #4  T=6.00000000e-06 (  6us). CC2500Net.Node2.cpu (id=21)
** Event #5  T=6.00000000e-06 (  6us). CC2500Net.Node2.gluelogic (id=24)
** Event #6  T=6.00000000e-06 (  6us). CC2500Net.Node2.cpu (id=21)
** Event #7  T=1.10000000e-05 ( 11us). CC2500Net.Node1.cpu (id=9)
** Event #8  T=1.10000000e-05 ( 11us). CC2500Net.Node1.gluelogic (id=12)
rf_test_ban() initialized
```

**Figure 8: Simulation with time annotation**

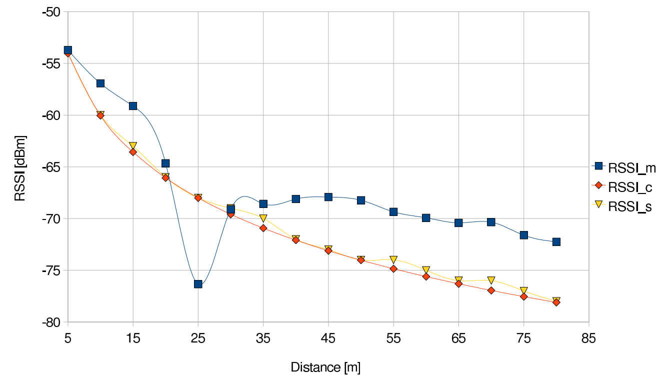


**Figure 9: Calculated, simulated and measured RSSI**

plained easily by the 2-ray model found in [10]. In addition to the *Line-of-sight* (LOS) path, the 2-ray model considers the wave reflection from the ground.

# 5. CONCLUSION

In this paper we have presented an approach to establish a power aware simulation of wireless sensor networks with nodes running real-life applications. Using a homogeneous simulation environment with only one simulation engine and time-annotated source code makes possible fast simulations at a high simulation accuracy. Separating between platform-dependent and platform-independent code as offered by the presented SW architecture allows easy porting to real sensor nodes. The concept was applied to a simple real-world networking test scenario and simulation results and real-world measurements were presented accordingly.

Future work will include the improvement of the PAWiS channel model to reflect large-scale and small-scale fading effects, the automation of time-annotation and the development of energy optimized protocols for real-time industrial applications using WSNs.

# 6. ACKNOWLEDGEMENTS

This work was funded by the COMET K2 Center "Austrian Center of Competence in Mechatronics (ACCM)". The COMET Program is funded by the Austrian Federal government, the Federal State Upper Austria and the Scientific Partners of ACCM.

*Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8643*
*http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8643*

## 7. REFERENCES


[1] antenova. *Titanis 2.4 GHz Swivel SMA Antenna*. http://www.antenova.com/?id=536.

[2] M. Bacivarov, S. Yoo, and A. Jerraya. Timed hw-sw cosimulation using native execution of os and application sw. In *High-Level Design Validation and Test Workshop, 2002. Seventh IEEE International*, pages 51–56, Oct. 2002.

[3] A. Bragagnini, F. Fummi, A. Huebner, G. Perbellini, and D. Quaglia. Co-simulation framework for the angel platform. In *Electronics, Circuits and Systems, 2007. ICECS 2007. 14th IEEE International Conference on*, pages 629–632, Dec. 2007.

[4] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Source-level execution time estimation of c programs. *Hardware/Software Co-Design, International Workshop on*, 0:98, 2001.

[5] F. Fummi, S. Martini, M. Monguzzi, G. Perbellini, and M. Poncino. Software/network co-simulation of heterogeneous industrial networks architectures. In *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pages 496–501, Oct. 2004.

[6] F. Fummi, M. Poncino, S. Martini, F. Ricciato, G. Perbellini, and M. Turolla. Heterogeneous co-simulation of networked embedded systems. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 3, pages 168–173, Feb. 2004.

[7] T. Instruments. *CC2500 Low-Cost Low-Power 2.4 GHz RF Transceiver*, swrs040b edition, September 2005. focus.ti.com/lit/ds/symlink/cc2500.pdf.

[8] J. Lessmann, P. Janacik, L. Lachev, and D. Orfanus. Comparative study of wireless network simulators. In *Networking, 2008. ICN 2008. Seventh International Conference on*, pages 517–523, April 2008.

[9] H. N. Pham, D. Pediaditakis, and A. Boulis. From simulation to real deployments in wsn and back. In *World of Wireless, Mobile and Multimedia Networks, 2007. WoWMoM 2007. IEEE International Symposium on a*, pages 1–6, June 2007.

[10] T. S. Rappaport. *Wireless Communications: Principles and Practice*. Prentice Hall PTR, 2nd edition, January 2002.

[11] A. Varga. The OMNeT++ discrete event simulation system. *Proceedings of the European Simulation Multiconference (ESM'2001)*, June 2001.

[12] D. Weber, J. Glaser, and S. Mahlknecht. Discrete event simulation framework for power aware wireless sensor networks. In *Industrial Informatics, 2007 5th IEEE International Conference on*, volume 1, pages 335–340, June 2007.

[13] X. Xian, W. Shi, and H. Huang. Comparison of OMNET++ and other simulator for wsn simulation. In *Industrial Electronics and Applications, 2008. ICIEA 2008. 3rd IEEE Conference on*, pages 1439–1443, June 2008.



*Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8643*
*http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8643*