Discrete Event Simulation Framework for Power Aware Wireless Sensor Networks

Daniel Weber, Johann Glaser and Stefan Mahlknecht

***Abstract*— The PAWiS (Power Aware Wireless Sensors) simulation framework facilitates design and simulation of wireless sensor network models. Main focus is given to power efficiency and therefore on capturing inefficiencies in various aspect of the system. These aspects include all layers of the communication system, the targeted class of application itself, the power supply and energy management, the central processing unit (CPU) and the sensor-actuator interface. The proposed simulation framework is based on the OMNeT++ discrete event simulator and provides PAWiS specific features (e.g. a mechanism to handle RF channel transmissions) to simulate, analyze and optimize the aforementioned aspects.**

I. Introduction

THE proposed simulation framework assists in developing, modeling, simulating, and optimizing wireless sensor network (WSN) nodes and network protocols. These sensor networks are mainly used in building automation, car interior devices, container tracking, building maintenance and monitoring, and geological surveillance. Individual nodes may comprise various types of sensors for temperature, humidity, insolation, strain gauge, and more. The simulation covers the internal structure of these nodes as well as communication among them. Sensor nodes forming a network communicate with each other via an ad-hoc multi-hop network (packets are routed from node to node whereas the routing information accumulates during the network operation).

The range of applications to be simulated covers simple applications like tiny sensor nodes (e.g. the TinyMote [1]), tire pressure monitoring, and car climate control as well as systems as complex as home entertainment systems (e.g. Sindrion).

Nevertheless, several aspects regarding power aware wireless sensors (PAWiS) are emphasized and directly supported by the proposed framework. The main goal is to simulate and develop PAWiS nodes in a way to reduce the overall power consumption by carefully optimizing various design aspects within the context of the application (e.g. nodes, potentially with a very low power consumption supplied from a single battery or utilizing energy scavenging to achieve lifetimes up to ten years).

II. Simulation Framework

A. General

Within the PAWiS simulation framework each node is built as a virtual prototype in a way that its function, timing and power consumption as well as system failures are simulated. The basic design methodology begins (according to a top down approach in Fig. 1) with a functional specification of nodes to specify node-internal interfaces, data flow and processes. After that model implementations have to meet architectural requirements stated by hard- and software constraints. These implementations are simulated and the resulting constraints in return affect the architecture and potentially the functional specifications in a bottom up manner.

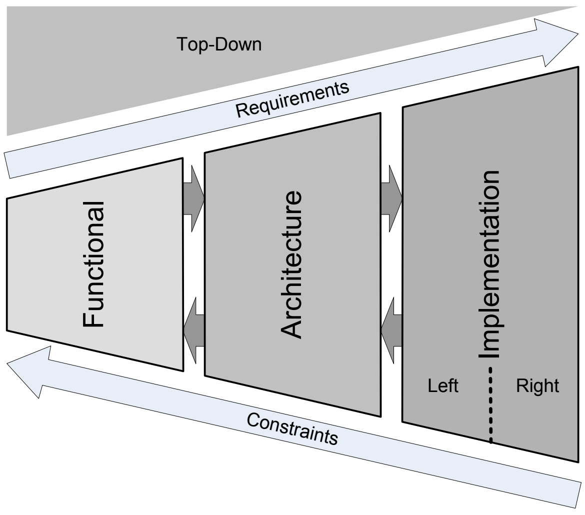

Fig. 1. Framework design methodology.

The framework is primarily focused on simulating inter- and intra-node communication. Additionally, finer grained aspects (e.g. CPU instruction set emulation as used by [6]) can be easily modeled with user extensions. However a trade-off between simulation details and execution performance (as discussed in [5]) has to be considered with increasing quantity of network nodes.

B. Structure

The PAWiS simulation framework is based on the OMNeT++ discrete event simulator [2] [3] which is written in the C++ programming language (Fig. 2). A discrete event simulation system operates on the basis of chronological consecutive events to change a system's state. These events are processed by the simulation kernel and can be further prioritized if more than one event occurs at an instant in simulation time to achieve a deterministic event execution. Simulation time itself does not progress in discrete steps but is advanced with each occurring event (hence it is not possible to issue events that are scheduled before the current simulation time). The proposed framework handles timing related issues with this discrete event mechanism.

The user of the framework is only confronted with OMNeT to comprehend the simulation process. User defined models are

Institute of Computer Technology, Technical University of Vienna
Email: {weber,glaser,mahlknecht}@ict.tuwien.ac.at

implemented in C++ and mostly utilize framework concepts. Node composition and network layout as well as environmental and setup parameters are specified in configuration files. User defined model implementations can be compiled (optionally with a GUI based on Tcl/Tk) to an executable simulator. The optional GUI enables visual debugging of the workflow and communication processes of the model on a per-event basis at simulation runtime. An additional feature is the

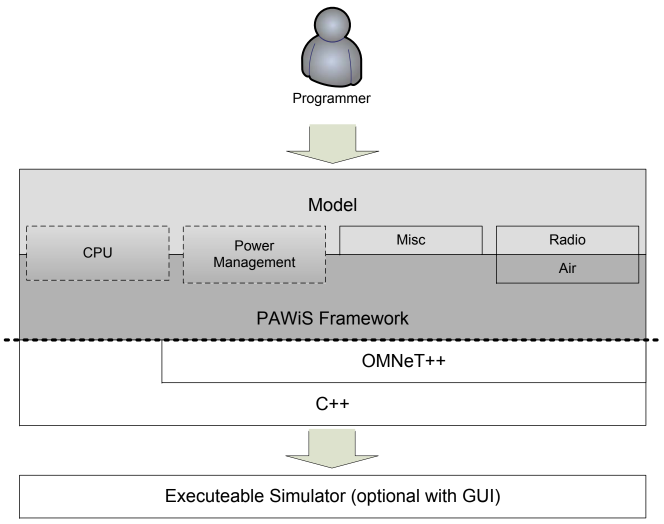

Fig. 2. Structure of the PAWiS simulation framework.

possibility to use SystemC [4] in parallel with OMNeT. This is achieved by modifying the OMNeT simulation kernel in a way that not only events from OMNeT are being processed but SystemC events are considered and processed at the appropriate point in simulation time. In the framework structure diagram (Fig. 2) the SystemC block would be located between the OMNeT++ block and the C++ block. That means the OMNeT kernel keeps the control and additionally dispatches SystemC events.

Primary simulation result represent logging information containing timing and power consumption profiles as well as event records. The completed model itself contains information regarding the functional description and architecture specifications along with low level implementation details.

C. Basic Concepts

1) Modularization: A wireless sensor node is typically composed of various hardware modules (e.g. a CPU, timer, radio and so on). The framework supports this composition of hardware modules and suggests an additional composition of software modules (e.g. the network protocol stack could be composed of modules representing application, routing, MAC and physical layer). The detail and granularity of sensor node decomposition strongly depends on the design and simulation requirements. If the main focus of the design is set to network layers it is recommended to provide a module for each layer in order to analyze multiple layer combinations. On the other hand a focus on hardware units would result in modules representing hardware or any applicable combination of software and hardware modules.

User modules (representing either hard- or software) simply need to be derived from a framework base class in order to be handled by the framework.

2) State machines: Every module is executing its specific jobs which can be represented as finite state machines (FSM), that are implemented as class methods. Within one module several FSMs can be implemented and run in parallel. The framework even allows to pass parameters to and from such tasks. The execution within one FSM is sequential but all FSMs are running apparently in concurrent fashion.

This concurrency is implemented as cooperative multi-threading. The framework provides yielding methods for FSMs where the execution control is handed back to the simulation kernel. This means that for a FSM, the program flow is suspended on this method (e.g. to wait for some condition to be satisfied) and will continue execution after being dispatched again afterwards. For FSMs simulation time usually elapses (or at least some other events occur) at these yielding methods.

3) Functional Interfaces: Control flow transitions between two modules are specified by so called *Functional Interfaces* (FI). They can be thought of as subroutines with well known names and parameter specifications. An invocation of a FI is similar to a blocking subroutine call but exceeds the module boundary. The framework allows the passing of arguments to and from FIs. In the model, FIs are implemented as class methods that are registered to the framework with their well known names. A collection of FIs grouped together under a well known name can be thought of as a functional module type description. This introduces a level of abstraction in the functional design process and enables reusability of functional design. Two modules that are completely satisfying the specification of a FI can said to be functional equivalent (although they might have entirely different power consumption and timing profiles).

4) CPU: Submodules of a sensor node are usually either implemented as firmware (software), i.e. are executed by a CPU, or as dedicated hardware. Simulation modules that are utilizing the CPU are referred to as software modules whereas other modules are referred to as hardware modules. It is important to note that software tasks (a task of a software module) can not run in parallel, since only one CPU is available. These software tasks give control to the CPU when they need to simulate code execution on the CPU. The CPU module of the framework ensures that only one task's code simulation is executed at a time and takes care of the task's timing behavior. When the CPU has finished simulation of a software task's code processing, it reactivates the task (for user this behaves like "call" and "return") which continues its execution after the CPU request.

To model the power consumption and timing behavior of software tasks the PAWiS simulation framework splits the simulation into two parts. The functional part of the software (the FSMs) is implemented in C++ inside the methods within a software module. The timing and power consumption part, on the other hand, is delegated to the CPU module which maintains its power consumption and delays execution of the software task for the calculated processing time (the time that the code execution on the CPU would take). The execution time and power consumption of a software task's code on the CPU is calculated in a user defined method and currently based on

- the percentage of integer operations,
- the percentage of floating point operations,
- the percentage of memory access operations,
- the percentage of flow control operations (loops, conditions) and
- the duration of code execution in norm time.

The *duration* is related to a *norm CPU* which means that the code in question would need *duration* seconds on an imaginary but well defined CPU. This abstraction of the CPU allows for a CPU exchange without the need to adapt other modules. The mapping from *norm CPU* time to real CPU time has to be provided by the user model.

5) Timing: Modeling time delays is different in firmware and hardware modules. For hardware modules the framework provides a simple wait method to suspend execution for a certain amount of time. A call to this method transfers execution control back to the simulation kernel which in turn dispatches other modules for execution. After the timeout has elapsed the module continues execution right after the call to the wait method.

However, this is not valid for software tasks because it is not possible to wait and do nothing in software (even an infinite loop without body does something on the CPU). In fact, if delays are needed in software the corresponding module has to use a loop (or a similar construct) to wait for a certain time and therefore utilize the CPU to achieve the timeout. The framework offers a variety of methods to utilize the CPU for timing and flow control purposes.

An important consequence of this timing model is that consecutive user code lines without a wait call or a CPU utilization request take place in the same simulation time instant (i.e. no simulation time elapses during that code execution). Simulation time only advances outside of user code (e.g. within the simulation kernel) or when these special methods are invoked.

6) Interrupts: The framework provides a basic mechanism to model interrupt handling which is implemented in several steps. It supports the handling in form of mappings between

- interrupt sources and interrupt vectors and
- interrupt vectors and interrupt service routines.

Additionally the framework takes care of the appropriate task scheduling when interrupts are issued.

Within the modeled microcontroller several interrupt sources exist (coming from other modules e.g. a timer, an analog-digital-converter,). Each of these sources is mapped to an interrupt vector. Furthermore every vector maintains a priority and an interrupt service routine (ISR). The framework's CPU module handles everything except prioritizing of interrupt vectors which has to be provided by the user model by overriding the CPU base class.

The user can register ISRs for interrupt vectors within software modules. When interrupt sources trigger interrupt requests, the CPU module determines the appropriate interrupt vector, checks its priority and if necessary transfers control to the ISR (which is within a software module). In case of a control transfer, the currently executed CPU task is interrupted and continues execution after the ISR has finished.

7) Environment: All sensor nodes are placed at 3D positions within the *Environment*. This is a representation of the the outer world and surroundings of all nodes including the RF channel.

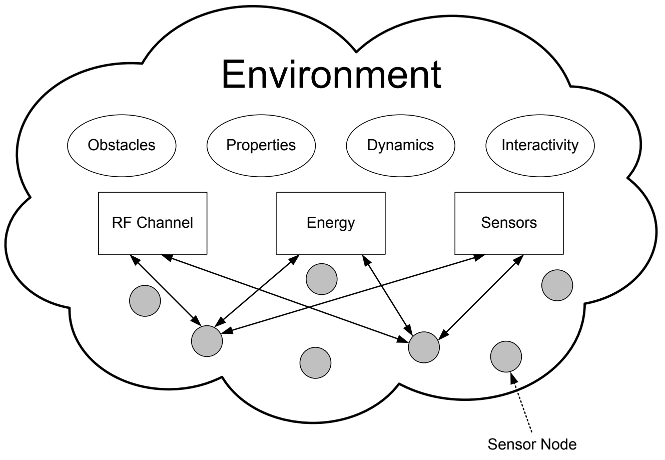

Fig. 3. The Environment with properties, objects and sensor nodes.

Besides the nodes themselves also other objects like walls, floors, trees, interferers, heaters, light sources, ... reside within the environment. Additionally global properties (e.g. the attenuation exponent b (see Sec. II-C.8)) are defined. The entire *Environment* can be configured with a config file.

The sensors of a WSN node can query physical environment data values from the *Environment* in order to simulate sensor readings. The values themselves could be simulated or acquired from a predefined data file. The framework user is totally free to set up the simulator for any physical unit required for his/her application.

From a software design point of view, the simulated sensor can register for notifications of changes in the targeted environmental data domain instead of frequently polling the domain thus reducing the computational load.

Future versions of the framework will include a script engine to simulate dynamic processes (e.g. simulate moving cars with built in sensor networks). Additionally a three dimensional environment visualization with interactive control abilities is planned.

8) Air: The *Air* is an essential part of the *Environment* to handle the RF channels, which are defined by node placement and obstacles between nodes. The RF signal is subject to wave propagation phenomenons like attenuation, reflection, refraction and fading (multi-path propagation) from the transmitter to the receiver.

a) Theory: The current implementation of the *Air* only handles attenuation effects, because other effects would require an enormous complexity in the simulator which is not necessary in the targeted field of application. For free space propagation the recipient power is defined by

$$\begin{aligned} P_{\mathrm{Rx}} &= P_{\mathrm{Tx}} G_{\mathrm{Tx}} \left(\overrightarrow{\varphi_{\mathrm{Tx,Rx}}}\right) \cdot \\ & \quad G_{\mathrm{Rx}} \left(\overrightarrow{\varphi_{\mathrm{Rx,Tx}}}\right) \left(\frac{\lambda}{4\pi}\right)^2 d^{-b} \end{aligned} \tag{1}$$

where $G_{\mathrm{Tx}} \left(\overrightarrow{\varphi_{\mathrm{Tx,Rx}}}\right)$ is the antenna gain of the transmitter in the direction to the receiver and $G_{\mathrm{Rx}} \left(\overrightarrow{\varphi_{\mathrm{Rx,Tx}}}\right)$ is the antenna gain of the receiver in the direction to the transmitter. λ is the wavelength, d is the distance between the two nodes and b is the attenuation exponent. The latter is typically around 2 for free space propagation, but for indoor environments higher values like 3.5 are more appropriate.

Alternatively the received power can be calculated by dividing the transmit power by the surface of a sphere $4\pi d^2$ (replacing the exponent by b) with radius equaling the distance between the receiver and transmitter d. After that the "diluted" power density is multiplied by the antenna area A_{Rx} of the receiver.

$$P_{\mathrm{Rx}} = P_{\mathrm{Tx}} \frac{1}{4\pi} d^{-b} A_{\mathrm{Rx}} \tag{2}$$

The attenuation from the transmitter j to receiver i is given by

$$A_{i,j} = \frac{P_{\mathrm{Rx},i}}{P_{\mathrm{Tx},j}} \tag{3}$$

where i and j are the index of the node ($i, j \in 1, 2, \ldots, n$). Calculating the attenuation from any node to every other node gives the node adjacency matrix

$$\underline{A} = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1} & A_{n,2} & \cdots & A_{n,n} \end{pmatrix}. \tag{4}$$

Let the vectors $\overrightarrow{P_{\mathrm{Tx}}}$ and $\overrightarrow{P_{\mathrm{Rx}}}$ hold the transmitted and received power of all n nodes, respectively. Then the calculation of received power is given by the simple matrix operation

$$\overrightarrow{P_{\mathrm{Rx}}} = \underline{A}\, \overrightarrow{P_{\mathrm{Tx}}}. \tag{5}$$

Since the results only make sense when a single transmitter is active, only one value in $\overrightarrow{P_{\mathrm{Tx}}}$ is different from zero and the matrix multiplication in (5) reduces to a single column scaling operation.

The adjacency matrix is a precisely defined interface from the *Environment* to the data communication. For an actual environment setup $\underline{A}$ is calculated. This is used during the network simulation as shown in (5). Due to this simple interface, $\underline{A}$ can also be calculated by an external RF channel simulation tool.

The current implementation of the *Air* supports only isotropic antennas with uniform antenna gain. Obstacles are considered for the adjacency matrix by explicitly given additional attenuation factors between pairs of nodes in the *Environment* configuration.

Whenever data packets are transmitted via the *Air*, every node with a received signal power above a certain threshold is notified. This notification contains the receiver power and the user data including the bit length of the transmission. Then the receiving node calculates the signal to noise ratio (SNR) between its own signal power P_{signal} and the received noise power P_{noise} with

$$\mathrm{SNR} = \frac{P_{\mathrm{signal}}}{P_{\mathrm{noise}}}. \tag{6}$$

From this SNR the bit error ratio (BER) is calculated (the formula can be provided by the user). The BER is a function of the SNR depending on the modulation format. From the calculated BER and the bit length of the transmission the bit error count is calculated and reported to the user module (e.g. to decide whether the received packet is valid or treated as noise).

If during the reception of a packet another node starts to send, this second signal is uncorrelated to the first sender. The framework models it as noise and therefore decreases the SNR of the receiver. Such events can happen several times during the reception of a data packet therefore the receiver has to deal with changing SNR throughout the packet receiving process. The final count of bit errors thus results from this sequence of different SNR values and is assembled from the portions of constant SNR. So the bit errors are calculated for all portions of the packet with constant SNR.

9) Power Simulation: A key feature of the framework (regarding PAWiS requirements) is given by the power consumption simulation of tasks. There are currently two ways to report power consumption requirement:

- directly report current and voltage or
- set up a hierarchy of power distribution and supply.

The first way is straightforward and just requires a function call to report the power requirement. This requirement is directly handed over to a central logging class called *Power Meter*. The resulting logging information is comprised of simulation time, voltage, current and the initiator of the task. With the initiator information it is possible to identify the module that requested the power consuming operation in the first place (e.g. the MAC module could be responsible for getting the radio module to consume some power).

The latter way of reporting relies on some setup to be done at simulation startup (the point where the network is generated). This setup includes registration of modules that serve as a power supply (it implements the appropriate interface) and subscribing tasks which is a method with a certain profile in any module that are utilizing such a power supply (Fig. 4 shows the relationship between a power supply and a subscriber called a power reporter as an UML model diagram). During the simulation a task calculates its current, reports this to its power supply and is notified about the actual input voltage by the power supply in return. This mechanism recursively propagates up the supply tree (breadth-first) and is finally reported to the *Power Meter*.

The consumed power equals $P = U \cdot I$ where the supply voltage U is provided by the supply module. Every subscriber (consumer or reporter) can have different electrical behavior, i.e. the current I depends in different ways on the supply voltage U. Currently three kinds of *Power Reporters* are provided but the framework facilitates the definition and usage of user defined ones.

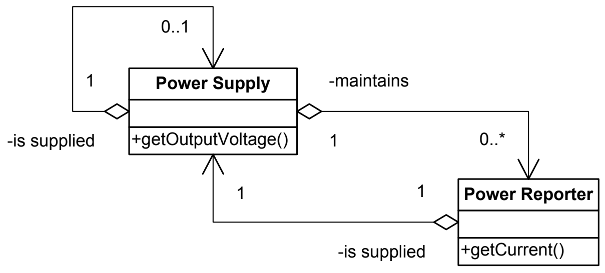

Fig. 4. Power simulation structure diagram.

- *Constant reporter*: $I = I_{\mathrm{const}}$. The current does not depend on the supply voltage.
- *Resistive reporter*: $I = \frac{U}{R}$ The supply current is proportional to the supply voltage, typically as for a resistor.
- *Linear reporter*: $I = I_{\mathrm{const}} + \frac{U}{R}$. A constant current plus a supply voltage dependant portion.
- *User defined reporter*: $I = f(U, ...)$. By deriving from a certain framework class it is possible to specify any electrical behavior (e.g. some non-linear characteristics).

A power supply module calculates its output voltage depending on the sum of output current reported by its subscribers, therefore the framework provides a mechanism to specify different power supply behaviors too. This power consumption model results in a simple electrical network.

A task does not necessarily need to keep a specified electrical behavior all the time. At any point in a task the power reporter characteristic can be changed resulting in an immediate update of the power supply. In return the new reporter handles all supply voltage changes according to its properties automatically.

Finally it has to be mentioned that only tasks that simulate dedicated hardware do direct power reporting as described above. On the other hand tasks that are intended to run as software just report its CPU utilization requirements. As a consequence the CPU calculates its own power consumption according to the utilization and reports this. The aforementioned initiator information of a power log entry is very important for software tasks, because the power consumer for a software task is always the CPU. Hence this information is needed to identify the specific task that caused the CPU to consume power.

III. Workflow

The basic idea of modeling a PAWiS node with the simulation framework is to decompose the node into functional blocks. These blocks can represent hardware as well as software depending on the project's requirements (e.g. application, physical, MAC, routing, timer, etc.). Each functional block can be implemented within a so called module to be accessible by the framework. Furthermore, various modules can be combined to form a node-module (called a compound module).

The implementation within a module has to meet the specification of the *Functional Interfaces* according to the type of the module. This forms another key idea of the modeling process, namely to provide a library with multiple different implementations for a specific type of module. The resulting module type implementation library can be used to evaluate, refine and test architectural issues of the user's model.

Initially a simulated PAWiS node may be composed of module type implementations that just meet the minimum functional requirements. However, it is important to note that modules, that represent two adjacent layers in the model design, do not necessarily need to have matching *Functional Interfaces*. So the model designer has to make sure that adjacent layer implementations meet each others functional interface requirements. After selecting the appropriate modules from the library and after configuring these modules (e.g. clock frequency of CPUs, resolution of ADCs, etc.) this basic model can be simulated. The results can now be reviewed to verify global or architectural model design issues. This concludes the initial phase in the design work flow.

A next step includes the refinement of the model or of specific modules. This can concern the model behavior (i.e. implement it in a more detailed and accurate way) as well as its function, module configuration or the entire node composition. After refining the model according to the project's design goals it can be simulated and the work flow cycle can start again.

This concludes the intended work flow which is a cyclic application of

- selecting module type implementations,
- configuration of modules,
- simulation of the model,
- evaluation of the model and
- refinement of the design.

These cyclic model improvements are called refinement cycles and are the main track to enhance the development and design [7]. When these refinement cycles are completed, the final outcome is comprised of

- the function,
- the architecture of the node,
- the implementation details, and
- the power specification of every module.

IV. Data Post Processing

During the simulation of a network model, a log file with information regarding

- power consumption,
- timing,
- events and
- module interaction.

is generated for further analysis. Although the resulting file is human readable it can be quite difficult to gain meaningful information from the plain text. In order to enable the user to easily extract useful information, an application is developed for displaying visual representations (Fig. 5 shows a screen shot of the application) of the log file (the application can be compiled under multiple platforms as it is written with wxWidgets [8] and OpenGL). The application parses the log file and organizes the information in a hierarchical manner (each of the following elements can also have events assigned) starting with

- PAWiS nodes,

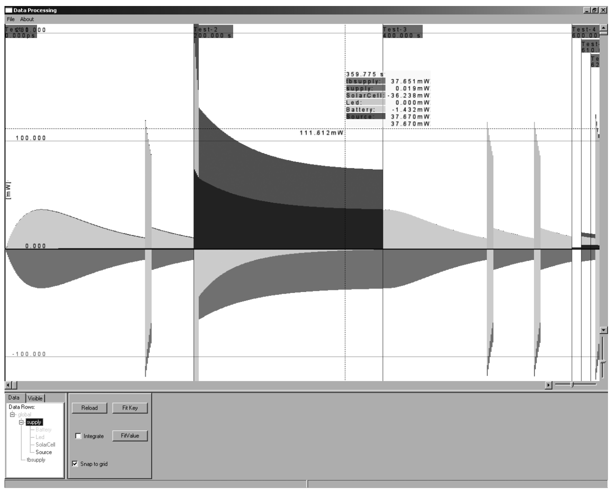

Fig. 5. Data post processing tool.

- modules,
- submodules and
- user defined categories.

Each of these elements can be enabled or disabled for displaying and are provided with a display color. Several navigational helpers like panning, zooming, scrolling, snapping and similar functionalities support the user. Additionally, the application provides various operations on data rows like integrating, finding significant power peaks, min-max and so on.

V. Optimization

Several strategies for the optimization of the wireless sensor system are available. The first alternative is to perform a *system level optimization*. This includes the node composition and even modifications of the whole system behavior like choosing different network layout or application patterns. System level optimization results in an adequate system architecture.

Alternatively *cross-layer optimization* can be performed. This means that more than one network layer is modified at a time. While it is possible that a single module update itself would degrade the node performance, the interaction of all module updates on the other hand leads to an improvement of the entire node performance.

A first technique to decide upon aspects for optimization of the model can be based on the pure function of the model (with the simulator GUI it is possible to see wrong or insufficient behavior during runtime). Additionally it is recommended to use the proposed data post processing tool to analyze the power consumption profile of nodes and distinct modules. Additionally the tool can be used to check the timing behavior and progression of certain events.

Within the framework, considering the aforementioned techniques, it is possible to use any form of optimization by applying the following strategies.

- Exchanging an actual module implementation for a certain module type (specified by its *Functional Interface*) by selecting a different one from the library, e.g. a dual-slope, a $\Sigma\Delta$ or an SAR ADC. Another example would be exchanging of communication layers (e.g. exchange the MAC protocol).
- Exchanging multiple module implementations for modules with *Functional Interfaces* that belong together (e.g. change MAC and routing protocol).
- Partitioning of modules and/or functions by dividing the task between hardware and software, digital and analog or RF and baseband. For example a specific MAC protocol could be implemented in software or as dedicated hardware acceleration unit. A combination of both is also possible.
- Re-scaling a module, e.g. the resolution of an ADC or the register count of a CPU.
- Parameterization of modules, e.g. the timing, transmission power and bit rate of a radio transmitter.

VI. Conclusion

The proposed PAWiS simulation framework is available in a preliminary version. It is currently being evaluated with the implementation of a model (including hard- and software) from a real sensor node [9].

OMNeT has turned out to be a good discrete event simulator with good support and a large community. The simulation framework model itself has been proven to be sufficient enough to meet the specified PAWiS domain requirements. However, various additional features will be added to later versions of the framework (e.g. dynamic simulation behavior via scripting, interactive 3D environment, more complex power handling methods, etc.).

Additional and updated information regarding the PAWiS project and the simulation framework as well as the framework source code can be found under [10].

References

[1] M. Roetzer, "Routing in energieautarken Funksensornetzwerken," Vienna, Austria, 2004.

[2] A. Varga, "The OMNeT++ discrete event simulation system," in *European Simulation Multiconference (ESM'2001)*, Prague, Czech Republic, 2001.

[3] "OMNeT++ Community Site," 2007. [Online]. Available: http://www.omnetpp.org

[4] "SystemC: Project Documentation," 2007. [Online]. Available: http://www.systemc.org

[5] J. Heidemann, N. Bulusu, J. Elson, C. Intanagonwiwat, K. chan Lan, Y. Xu, W. Ye, D. Estrin, and R. Govindan, "Effects of detail in wireless network simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, USC/Information Sciences Institute. Phoenix, Arizona, USA: Society for Computer Simulation, January 2001, pp. 3–11.

[6] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J. S. Baras, "Atemu: A fine-grained sensor network simulator," in *First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, 4.-7. Oct. 2004, pp. 145–152.

[7] S. Mahlknecht, J. Glaser, and T. Herndl, "PAWIS: Towards a Power Aware System Architecture for a SoC/SiP Wireless Sensor and Actor Node Implementation," in *Proceedings of 6th IFAC International Conference on Fieldbus Systems and their Applications*, Puebla, Mexiko, 14.-15. Nov. 2005, pp. 129–134.

[8] "wxWidgets Homepage," 2007. [Online]. Available: http://www.wxwidgets.org/

[9] S. Mahlknecht, S. A. Madani, and M. Roetzer, "Energy Aware Distance Vector Routing Scheme for Data Centric Low Power Wireless Sensor Networks," in *4th International IEEE Conference on Industrial Informatics INDIN 06*, 16-18 Aug. 2006.

[10] "PAWiS Project Homepage," 2007. [Online]. Available: http://www.ict.tuwien.ac.at/pawis/