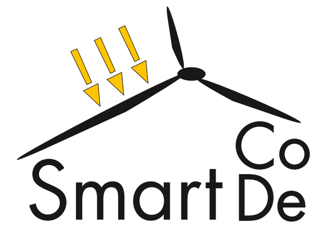
**European Commission**

**Information Society and Media Directorate - General**




# Executable specification of a SmartCoDe node

SmartCoDe

| | |
|---|---|
| Project No.: | ICT-2009-247473 |
| Deliverable No.: | D-2.2 |
| Deliverable Title: | Executable specification of a Smart-CoDe node |
| Due Date: | December 31th 2010 |

| | |
|---|---|
| Nature: | Prototype |
| Dissemination Level: | public |
| Authors: | Stefan Mahlknecht, Markus Damm, Javier Moreno, Edgar Holleis |
| Lead Beneficiary No.: | 3 |
| Lead Beneficiary: | TUV |

SmartCoDe has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 247473

# Contents


1 **Introduction** 3

2 **The SmartCoDe node functional model** 4
- 2.1 SmartCoDe Simulation Framework: General Approach . . . . . 5
- 2.2 Sensor and Actuator Models . . . . . 5
- 2.3 Physical EuP simulation . . . . . 6

3 **Network Simulation** 8
- 3.1 Previous Work . . . . . 8
- 3.2 Physical Layer Model . . . . . 9

4 **Implementation of the SmartCoDe Functional Model** 10
- 4.1 Smart Node Base Class . . . . . 10
- 4.2 Events . . . . . 12
- 4.3 Sensors and Actuators . . . . . 12
- 4.4 Top Level API . . . . . 13

5 **Example** 15

6 **Conclusion and future work** 17

**References** 18

**Abbreviations and Definitions** 18

# 1 Introduction

This report documents the modelling efforts in Task 2.3 for the system design of a SmartCoDe node. A current version of the modelling and simulation environment with simulation examples can be delivered to the commission upon request. Since the simulation environment consists of several components, we provide a virtual image of a Linux environment (Ubuntu) for VirtualBox (or VMWare if preferred) having the simulation environment already installed.

The simulation environment is based on the C++ System modelling and simulation framework SystemC [9]. It extends the C++ languages by primitives for system modeling: concurrency, signals, events, handling of simulation time and a simulation kernel. The SmartCoDe simulation framework also makes use of the SystemC extension libraries TLM 2.0 [1], which targets abstraction of communication, and SystemC-AMS 1.0 [10] for analogue and mixed-signal modelling. The wireless network simulation is based on the SNOPS simulation framework [2]. Figure 1 summarizes the architecture of the simulation stack.

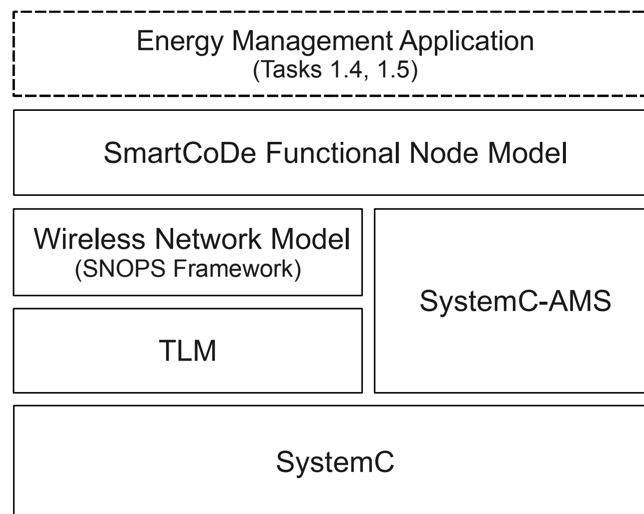



Figure 1: SmartCoDe Simulation Architecture

The development of the energy management application (the top layer in figure 1) is the focus of the tasks 1.4 and 1.5. For the purpose of this deliverable it exists only as proof-of-concept programming example of how to make use the SmartCoDe simulation framework.

This work builds upon the efforts undertaken in work package 1, especially the classification of energy using products (EuPs). The general idea is to base the node design on this classification. On the hardware side, this could lead to different design variants depending on the EuP class, e.g. lighting applications (class VARSVC) could be handled by simpler nodes, although the target is to provide a hardware platform which is suitable for all EuP classes. In either case, the embedded software will be different for each class in order to provide for each EuP class a matching energy management strategy.

**It is recommended to read Deliverable D-1.1.1 before reading this report**, since some concepts introduced there are referred to in this document. This applies not only to the EuP classification, but also the decentralised energy management approach.

A consequence of D-1.1.1 is that the SmartCoDe project needs a model which can be used to develop a decentralised energy management algorithm as a basis for Task 1.4 starting in January 2011. A standalone model of a SmartCoDe node is not sufficient, the whole communication infrastructure and network needs to be modelled. At the node level, a high level API and a functional model are sufficient.

The rest of the document is structured as follows: Section 2 discusses the functional model of the SmartCoDe node, section 3 the network simulation framework, section 4 the implementation and section 5 discusses an example energy management algorithm.

# 2 The SmartCoDe node functional model

There are three use cases for the SmartCoDe node functional model:

- Hardware / Software Co-Development
- Validating distributed energy management algorithms in within its communication context
- Prototyping SmartCoDe node / EuP integration

For these purposes the SmartCoDe simulation framework models the relevant parts of the hardware (wireless transceiver, sensor and actuator-interfaces) as well as the software (embedded RTOS). Hardware components are modelled using SystemC, SystemC TLM and SystemC-AMS. The software APIs presented to application level code functionally resemble JenOS, the ZigBee SoC chosen for the SmartCoDe demonstrator.

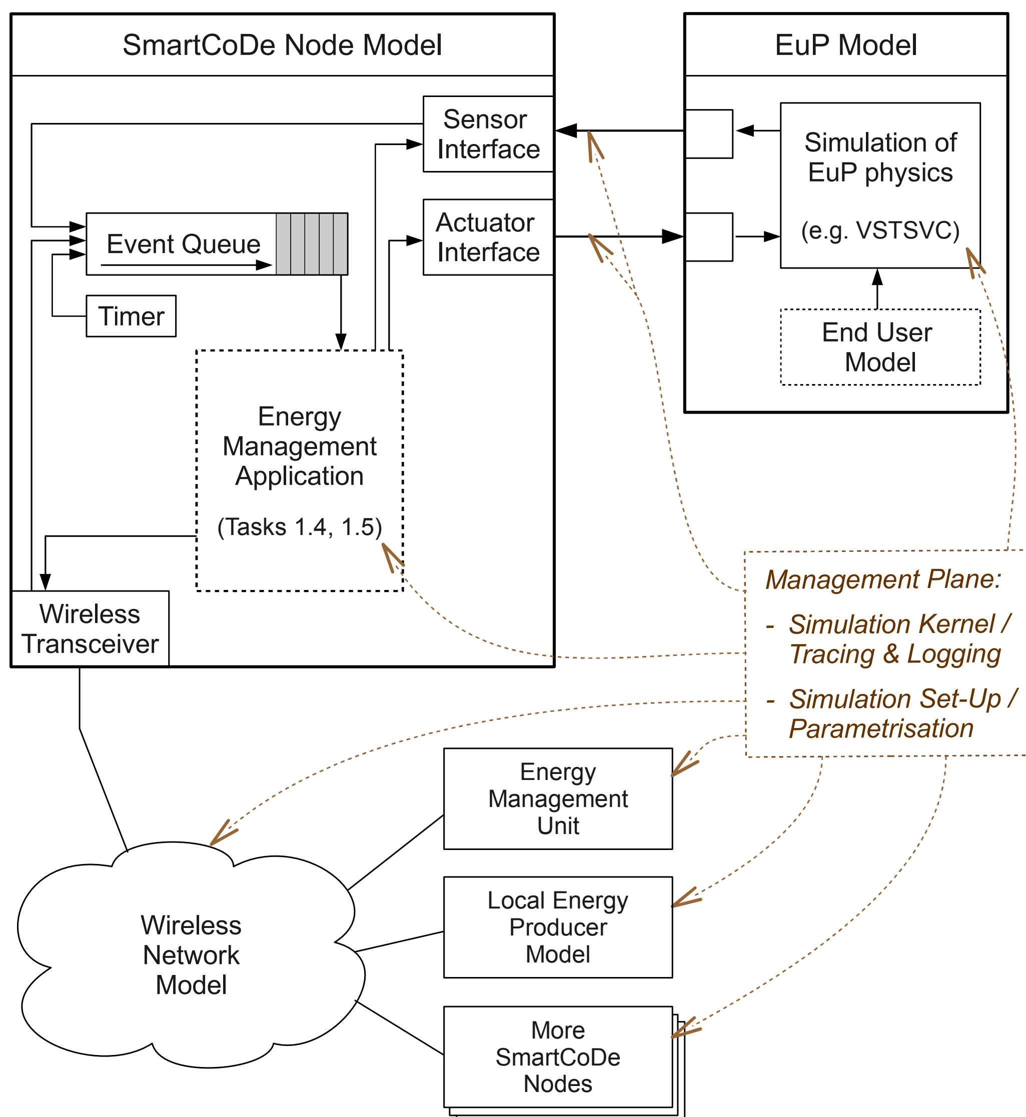



Figure 2: Architecture of the SmartCoDe functional node model embedded into the SmartCoDe simulation framework.

Figure 2 reveals the inner details of the SmartCoDe functional node model and how it interacts with the EuP and the wireless network within the simulation framework. The different elements are described in more detail in the following sections:

- SmartCoDe functional node model → sections 2.1 and 2.2
- EuP model → section 2.3
- Wireless network model → section 3

## 2.1 SmartCoDe Simulation Framework: General Approach

In order to facilitate later porting of application code from the simulation framework to the SmartCoDe demonstrator, the APIs between virtual SmartCoDe node and application code resemble the concrete environment found inside the NXP/Jennic platform chosen for the SmartCoDe demonstrator. That means that the offered operating system primitives are similar, not the concrete formulation of the API calls.

The energy management application to be simulated in the SmartCoDe simulation framework is written in an event-driven style. This reflects common practice in embedded system programming. Even though the NXP/Jennic platform does support multi-threading, the recommended way of writing applications is to use only a small number of threads, usually just one, for application level code. Other threads handle network communication tasks, network originated remote procedure calls (RPCs), as well as hardware related tasks. True to the TLM paradigm, those other tasks are not modelled in detail (as would be the case in hardware oriented modelling), but merely their latencies are accounted for. The OS model for SmartCoDe node functional model can therefore forego true multi-tasking. Instead, there exists a single thread for the application level code which is driven by a single unified event queue.

The downside of this approach is that CPU utilisation and especially contention is not accurately represented by the model. It was deemed of low priority for the SmartCoDe demonstration, because the selected hardware platform is assumed to provide ample reserve in that respect. For this deliverable, it was instead decided to focus on networking aspects (bandwidth and latency of the wireless channel), as well as energy management aspects - providing a virtual platform for developing and proofing energy management algorithms. Should the need arise later in the project, the issue will be revisited.

As depicted in Figure 2, the unified event queue is fed by the following event sources:

- Network events: Incoming packets, changed network variables
- Sensor events: Whenever new sensor values become available
- User interface events: Modelled as sensor events
- Timer events: For delayed or periodic execution

Low level networking tasks (ZigBee NWK and below) like routing and forwarding packets, maintenance of tables and channel management are not exposed to the application level code.

Following the example of ZigBee, the SmartCoDe nodes do not currently handle wall-clock time. Timed sequences are driven by the device-local clock, network interchanges are driven by relative time. This assumption has consequences for the maximum allowable lifetime of network packets (how often and how long they may be buffered), the minimum duty cycle of power cycling devices and the convergence rate of distributed energy management algorithms. The issue may be revisited at a later time, if justified by results of the energy management simulation.

## 2.2 Sensor and Actuator Models

Three different kinds of sensors are supported by the sensor interface:

- **Periodically triggered sensors** - periodically write sensor values into the event queue (the interval is sensor specific)
- **Single shot sensors** - when activated, write exactly one sensor value to the event queue (after exhibiting a sensor specific delay)

- **Externally triggered sensors** - triggered by events from the physical domain, writes one sensor value to the event queue

Common to all three sensor types are the following characteristics:

- Sensor values are represented as unsigned 16 bit integers
- The most recent sensor value (along with its age) is cached and can be queried without delay
- Sensors can be turned on and off (irrelevant for single shot sensors)
- If relevant, the sensor's semantics can be declared by specifying ZigBee Cluster descriptors
- ZigBee-compatible sensor values can be sent to and received from remote nodes (using network variables)

The interface is designed to support all sensors present in building automation, as well as some specific sensors inside white box appliances with relevance to their function as virtual energy storages. What is explicitly not abstracted is the mapping formula from physical dimension to sensor value. The ZigBee standard is taken as guideline. In other cases the energy management application will need to handle the mapping by itself.

User interface controls (which allow the consumer to control the EuP) are captured by the sensor interface as well. Push buttons are implemented as a sensor that writes a "1"-value to the event queue. Debouncing is not necessary within the simulation. A hand wheel or slider for adjusting a level is modelled as being connected to an analogue input which is periodically sampled. It is not intended to model complex user interactions in the SmartCoDe node functional model. If necessary they can, however, be modelled in the EuP connected to the SmartCoDe node. In that case it is the interface between EuP and SmartCoDe node that is captured by the sensor & actuator interface.

The actuator interface is even simpler, it is modelled after a single analogue output, represented by unsigned 8 bit integer. That should be sufficient for the purpose of simulating energy management applications.

## 2.3 Physical EuP simulation

According to the classification of EuPs (as described in deliverable D1.1.1), the EuPs need to be simulated in a level of detail adequate for calculating their energy consumption. The simulation framework contains for each identified class an abstract implementation that can then be further specialized into concrete devices. Device usage can be either deterministic for optimizing the energy management algorithm for certain scenarios, or the device usage patterns can be fed by a statistical model provisioned from publicly available data-sources of typical energy usage, such as the REMODECE project(see [3]).

Besides network communication and energy consumption, the following points represent further quality considerations (metrics) for candidate energy management algorithms:

- Additional fatigue on devices which may be designed to endure only a limited number of power cycles
- Conformance to consumer expectations, or how often consumers may be compelled to make use of '*override*' facilities because of dissatisfaction with default energy management policies

EuPs are connected to the SmartCoDe nodes via the sensor and actuator interfaces; they have to implement the secondary side of these interfaces. For most device classes, the actual physics inside the devices are of no interest to the simulation framework. Power profile, communication dependencies and usage patterns sufficiently abstract what the energy management unit needs to know about a particular device.

This is not true for virtual storage devices (class VSTSVC). For the EuPs we consider in SmartCoDe, the VSTSVC-class will mostly contain devices providing a thermal service (heating, cooling). The respective thermal process (e.g. room temperature or the temperature inside a fridge) is modelled in SystemC-AMS.

The simulation framework already contains an exemplary model of a VSTSVC-class device that can be parametrised to function either as an electrical heating or cooling device.

SystemC-AMS is used for modelling the temperature since it provides the capabilities to model such continuous processes in discrete time. At TUV, there is prior experience in modelling thermal behaviour using a simplified pseudo-electro-statically equivalent approach, based on low-pass filters [7]. The mechanism is depicted in figure 3.

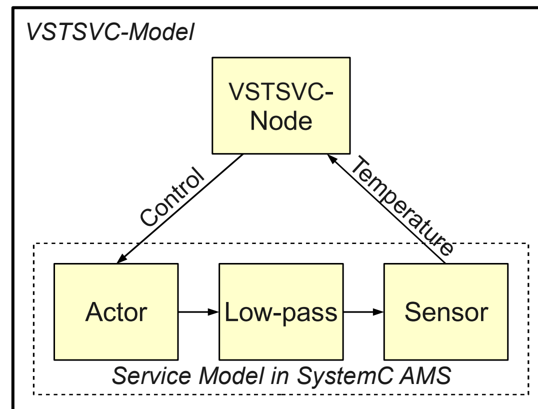



Figure 3: Model of an Virtually Storable Service (VSTSVC) EuP in SystemC

Out of scope of the physical simulation is a detailed thermal model of buildings. The SmartCoDe project focuses its efforts on management of electrical energy usage in homes and offices, not on thermal management. However, in principle the physical models in SystemC-AMS could be coupled, e.g. to model neighbouring rooms and the interdependence of their respective temperatures.

A crucial aspect is the effect of consumer behaviour. For a realistic simulation, consumer behaviour has to be incorporated into the simulation, e.g. switching of lights, but also influencing the thermal processes in the model indirectly (e.g. opening window). In how far and how detailed consumer behaviour has to be modelled within SmartCoDe will be determined within work packages 1.4 and 1.5.

# 3 Network Simulation

The SmartCoDe node functional model recreates select features of the ZigBee [13] and IEEE 802.15.4 [6] stacks. The main focus is:

- Realistic handling of broadcast messages
- Realistic handling of passive delivery to power-cycling devices
- Support for network variables (ZigBee cluster attributes)

IEEE 802.15.4 [6] is a low power, low complexity wireless networking protocol. One of its core assumptions is that the duty cycle of the channel is but 1%. The challenge for the networking simulation is to verify that the application uses the network in a way compatible with the design assumptions of the underlying networking technology. Not in scope of the SmartCoDe node functional model is a realistic implementation of the routing protocol, because priority is given to the simulation of completely commissioned networks where all routes have already been established. The commissioning process, including key-establishment, will be handled at a later stage.

Realistic handling of broadcast messages involves maintenance of broadcast transaction records as described in the ZigBee standard [13]. With respect to TLM-style handling of broadcast messages, the radius field (together with other MAC and NWK level headers) will be handled as a generic payload extension to TLM's built-in facilities. The desired outcome is that the relative complexity of broadcast handling does not translate to high simulation overhead. The generic payload is instantiated only once and propagated (rebroadcast) throughout the network without excessive data copying within the simulation framework. Bandwidth costs and effects on noise are accounted for nonetheless.

Passive delivery is the ability for end devices to turn off their receiver most of the time and only periodically poll their router. The mechanism is specified in [6]. Power cycling end devices periodically send a MAC-level data-request command to their ZigBee router which has to answer within a predetermined time during which the end device keeps its receiver on. From a TLM perspective, the data-request command message can be heavily reused since it lives only for a very short time, whereas the lifetime of regular messages is extended until they are either delivered to all end devices or expire in the router's buffer.

Network variables, finally, are a convenient ZigBee mechanism to make the network transparent to the application. The application can be written in ignorance of where it gets its sensor data from, i.e. which other nodes it is *bound* to. Network variables can be read and written to, or the remote node can be configured to report attribute changes automatically. For the simulation it means that binding information is read from the configuration file and the network is considered readily configured at simulation startup. Propagation of sensor data is then handled by the virtual ZigBee stack and does not have to be explicitly coded in the energy management application.

The physical layer radio model is crucial to evaluate the system. Through an environment model, all propagation effects can be taken into account. As a result, a realistic communication simulation can be used for validating algorithms and approaches.

## 3.1 Previous Work

The radio model is based on previous work developed as part of the PAWiS (Power Aware Wireless Sensors) and SNOPS (Sensor Networks Optimization by Power Simulation) projects.

#### Discrete Event Simulation of Wireless Sensor Networks: PAWiS Framework

During the PAWiS project [11], a framework for modelling ultra low power sensor networks was developed [4]. The PAWiS Framework is based on OMNeT++ [12] discrete event simulator, oriented primarily for building network simulators. The main purpose of the simulation was to model power and energy consumption to estimate the network lifetime.

PAWiS Framework included a radio and propagation model, capable of evaluating attenuation, collisions, noise and interferences. Every transmission in the network was transformed into as many point to point

messages as neighbour nodes within the range. PAWiS environment model is passive and only tells the nodes which nodes receive their messages and with how much attenuation.

OMNeT++ is a network oriented simulator, based on C++. Hardware models have to be written in C/C++. Otherwise, co-simulation is required.

Hardware models in PAWiS, especially microcontroller model, did not have the accuracy necessary for estimating power consumption, which requires some knowledge about the time required by the processor to finish its tasks. Instruction Set Simulators (ISS) or cycle-accurate models are needed to precisely determine this.

#### Transaction Level Modelling of Wireless Sensor Networks: SNOPS Framework

Transaction Level Modeling (TLM) is a high-level approach for modelling digital systems [8]. Communication is abstracted into transactions. This abstraction pursues not only a performance boost but improving the interoperability as well. Transactions are requested through interfaces, and go in and out of the modules through the corresponding sockets. Hence, module implementation is kept independent of the communication model.

TLM is however, originally oriented for bus communication abstraction, which is a priori completely different from wireless communication. Nonetheless, a correspondence can be established between the elements of a wireless communication and TLM basic concepts [5].

The framework developed during SNOPS project is based on the PAWiS Framework. Nevertheless, OMNeT++ functionality has been completely replaced by SystemC [9] and TLM.

The environment model is very similar to the PAWiS model. However, in the SNOPS case, the environment takes part in the communication indeed, by actively distributing messages from sender to receiver nodes.

Unlike in PAWiS, in the SNOPS Framework there is only one physical copy of the message for every complete end-to-end message delivery, including all intermediate hops. This implies a significant improvement in terms of simulation performance [2]. Furthermore, as OMNeT++ simulation core is replaced by SystemC, SystemC or even SystemC-AMS system models can be directly integrated in the simulation.

In the course of the SmartCoDe project, the SNOPS Framework is extended: to efficiently handle broadcasting custom SmartCoDe fields are added to the internal SNOPS generic payload extension. This is advantageous, compared to implementing new generic payload extensions which in case of broadcasting would grow hugely in a short time interval. The new features are contributed back to SNOPS software increasing its robustness and supporting future applications.

## 3.2 Physical Layer Model

SmartCoDe's physical layer model of the network is built upon the SNOPS Framework. The main characteristics of the SmartCoDe simulation framework's physical layer model are described below.

- Models a generic transceiver which lets the simulation engineer choose a wide range of parameters: Modulation, data rate, frequency bands
- Estimates transmission ranges based on attenuation, which depends on the distance between nodes. Attenuation formula can be configured according to the scenario, either by setting the corresponding attenuation exponent or by providing a custom adjacency matrix calculation method. Additional attenuations, such as those due to obstacles can be also be implemented.
- Models collisions and noise from other transmitting nodes, as well as interferences coming from other communication channels.
- Node displacement, joining and leaving automatically trigger a recalculation of the attenuation matrix. Other time-variant effects can be taken into account as well by implementing additional attenuations.

The implemented features comprise a comprehensive radio model and form a robust basis for tasks 1.4 and 1.5 – development of the SmartCoDe distributed energy management algorithm.

# 4 Implementation of the SmartCoDe Functional Model

Figure 4, is a UML diagram of the SmartCoDe functional node model. The classes and its members are described below.

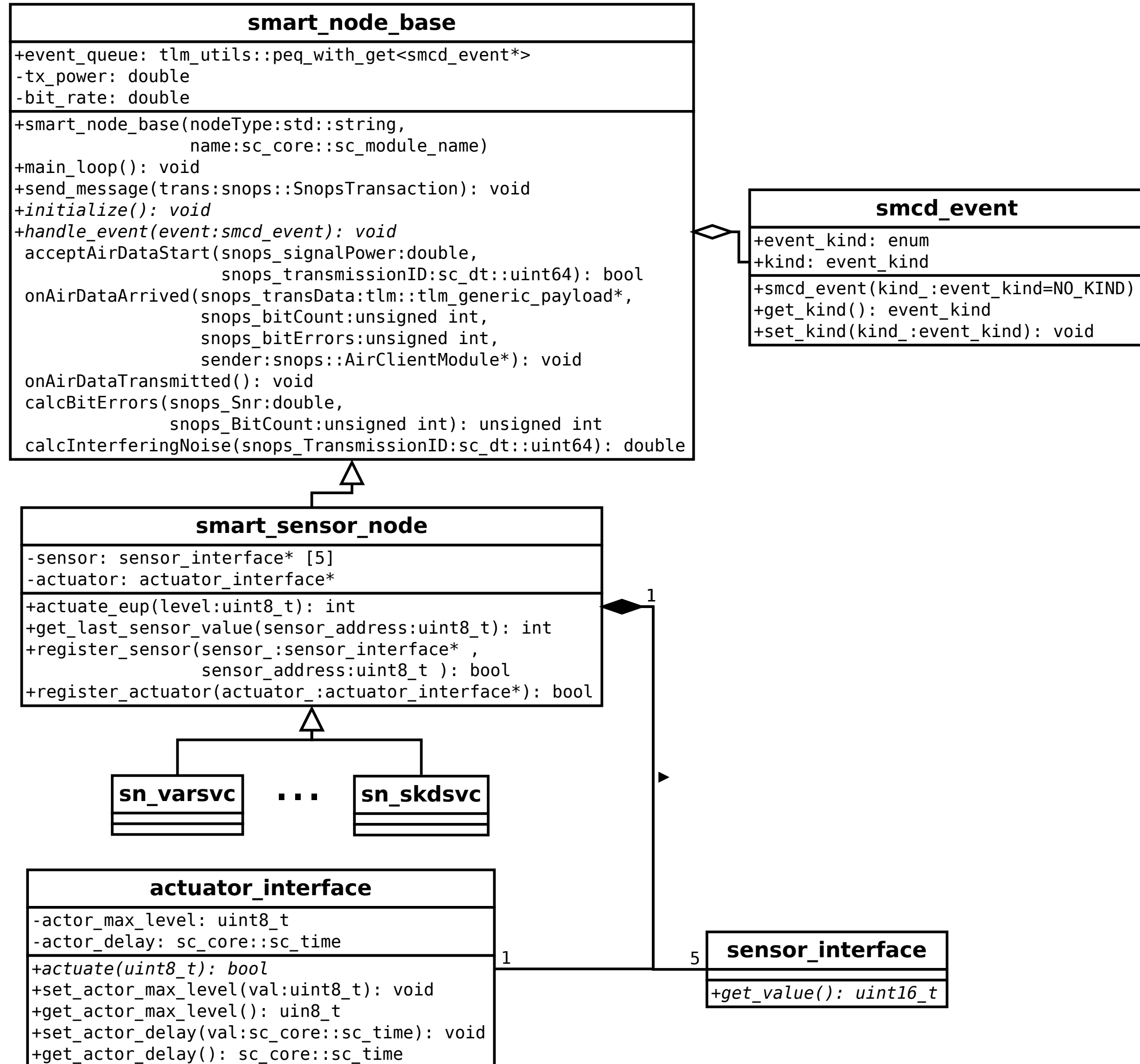



Figure 4: Inheritance and composition relationships within the SmartCoDe simulation framework. The energy management algorithm is to be implemented by overriding function *handle_event* in the subclasses of *smart_sensor_node*.

## 4.1 Smart Node Base Class

Class *smart_node_base* serves a dual purpose: It interfaces the SmartCoDe simulation framework to the SNOPS wireless simulation framework thereby serves as the base class for all SmartCoDe functional node models. All parameters, virtual methods and callbacks required by the SNOPS framework are implemented, and the functionality of participating in wireless networking simulation is offered to derived classes. In Figure 5, attributes and methods of the *smart_node_base* class are shown.

```
smart_node_base
+event_queue: tlm_utils::peq_with_get<smcd_event*>
-tx_power: double
-bit_rate: double
+smart_node_base(nodeType:std::string,
                 name:sc_core::sc_module_name)
+main_loop(): void
+send_message(trans:snops::SnopsTransaction): void
+initialize(): void
+handle_event(event:smcd_event): void
 acceptAirDataStart(snops_signalPower:double,
                    snops_transmissionID:sc_dt::uint64): bool
 onAirDataArrived(snops_transData:tlm::tlm_generic_payload*,
                  snops_bitCount:unsigned int,
                  snops_bitErrors:unsigned int,
                  sender:snops::AirClientModule*): void
 onAirDataTransmitted(): void
 calcBitErrors(snops_Snr:double,
               snops_BitCount:unsigned int): unsigned int
 calcInterferingNoise(snops_TransmissionID:sc_dt::uint64): double
```

Figure 5: The SmartCoDe functional node's base class

It is assumed that every node in the simulation is connected to the network via wireless and maintains its own event queue. Hence, this is exactly what the smart node base class provides.

In order to communicate through the SNOPS Framework, the module inherits from SNOPS AirClient-Module class and implement five virtual methods:

- **acceptAirDataStart** - Entrance point for received messages: if transceiver is in listening mode, available, and received signal power is over the receiver sensitivity threshold, the message can be received. Otherwise, it is ruled out and considered as noise.
- **onAirDataArrived** - Once accepted, the reception process starts. This method is called when reception is completed and data is available. In the SmartCoDe simulation, when a message is received, it is encapsulated into an event and added as an untimed event to the event queue.
- **onAirDataTransmitted** - Returns control to the node once the transmission to the air is finished.
- **calcBitErrors** - Polymorphism is required here, as bit errors depend on the modulation, which is defined by the simulation engineer. As the NXP/Jennic transceiver operates in the 2.4 GHz frequency band, the implementation of this method corresponds to the only modulation defined by IEEE 802.15.4 Standard [6] at this frequency, which is offset quadrature phase shift keying (O-QPSK).
- **calcInterferringNoise** - Models noise coming from other channels. In SmartCoDe, all signals are transmitted through the same channel, so the ideal case of Adjacent Channel Rejection Ratio (ACRR) is considered.

In order to support different kind of devices and different event handling, two virtual methods are defined:

- **initialize** - All initialization data shall be set up here. In addition, the first event, if any, is added to the queue at the corresponding time.
- **handle_event** - Called whenever an event is collected. Depending on the kind of the event, the appropriate action is started.

The following two methods are also provided by this base class.

- **main_loop** - Is in charge of collecting events and calling the handle_event method. When there are no more events at the current time, it also *waits* till the next instant in which there is a new event, i.e. advances the simulation time.
- **send_message** - sends the message to the air with the corresponding transmission parameters, such as the transmitting power or the transmission delay. These parameters are defined by the transceiver used. NXP/Jennic values are used by default.

## 4.2 Events

As stated in Section 2.1, the simulation will follow an event- driven paradigm. These events are appended to a unified event queue, provided by TLM. It permits adding timed and untimed events to the queue. Whenever the last untimed event is collected, the clock is advanced to the time of the next event.

Events are modelled as subclass of the TLM generic payload, they therefore provide a data pointer that supplies the associated information. Since messages or sensor received values are also modelled as events, they must contain a pointer to the message or value received

In view of the fact that the event queue comprises all kind of events, which trigger completely different tasks, a variable containing the kind of the event is required. Therefore, several type of events have been implemented.

- **Periodic event** - Nodes are usually duty-cycled. Thus, this event triggers the beginning of the active state. In the handle_event method, the next periodic event, at the corresponding time must be added to the queue.
- **Message arrival event** - The message is attached to the event in the transaction data pointer.
- **Schedule event** - Use case: schedule turning on or off the device depending on the forecast arrived.
- **Sensor event** - Some sensors send information to the node at specific times. Enclosed in event's data pointer must be the value sensed.

These types are not fixed and can be extended depending on the requirements.

## 4.3 Sensors and Actuators

Sensors and actuators are dependant on the connected EuP. However, a common interface has been defined for two main reasons: Firstly because both subsystems are not accessed directly, but from the base class; and secondly because this implementation aims to provide a uniform top level API to the simulation engineer.

There are two implementation aspects related to sensors and actuators. The first one deals with the implementation of the physics measured by these sensors. The second one is how the information between the sensor and the node, or the node and the actuator, is exchanged. In Figure 6, both aspects and the way they are connected to the framework are described.

In the first case, physics of the sensed values and implementation details of sensors and actuators, must be implemented as a subclass of sensor interface and actuator interface classes, respectively. Then, virtual methods get_value and actuate, have to be implemented.

Using SystemC as the simulation kernel permits modelling the physics as analogue systems using SystemC-AMS.

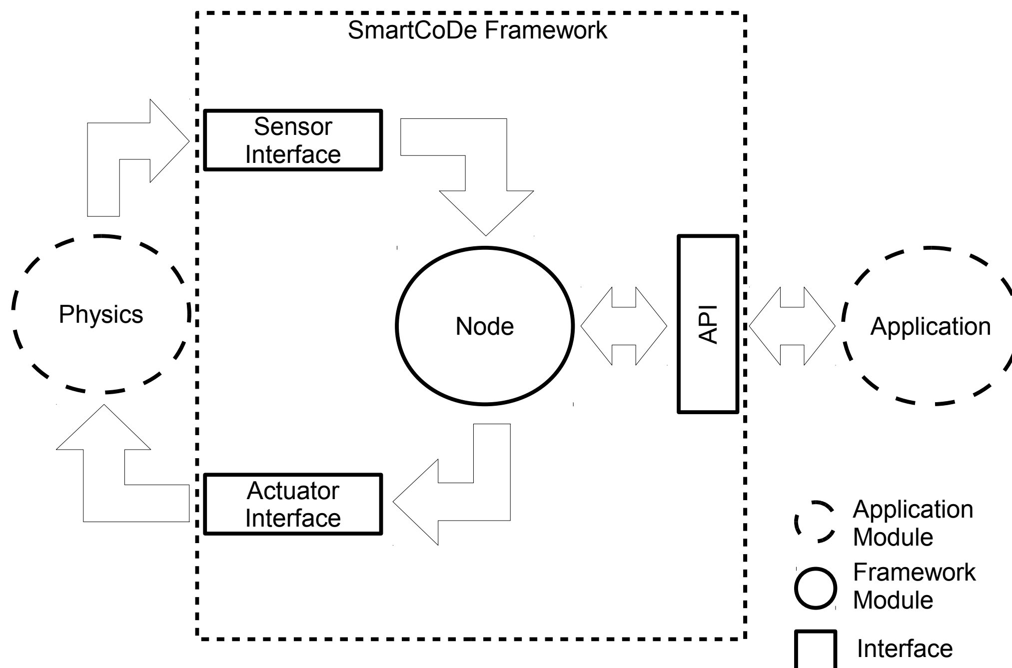



Figure 6: Architecture of the two implementation aspects concerning sensors and actuators

In order to model the power consumption, state machines provided by the SNOPS Framework are used. These state machines are built in such a way that only possible transitions between states are allowed. Hence, once the state machine is configured, there is no possibility of triggering an erroneous state transition, which is usually a source of errors very difficult to detect. Every state has an associated power consumption value. Thus, power consumption can be obtained by maintaining a finite state machine of the device and requesting the power consumption value at the current state.

Every sensor node must be a subclass of the smart_sensor_node class. Each smart sensor node contains one actuator and three sensors (power, temperature, and specific sensor). When a device is implemented, actuators and sensors must be registered at the node. In the case of the sensor, as there are several, an address parameter must be provided.

As long as the sensors and the actuator have been registered in the node, they can all be accessed through the interfaces provided to this end: actuate_eup and get_last_sensor_value.

## 4.4 Top Level API

This section summarizes the implementation such that only functions that concern the simulation engineer are described.

Table 1 depicts all functions provided by node parent classes that are used by the energy management application in order to configure and interact with the low level features.

The Application Programming Interface (API) provided can be split into two main aspects: Setting up or configuration of the simulation and interaction with lower level functionalities.

### Configuration

In order to set up a simulation the programmer has to define his devices. Sensor nodes must inherit from smart_sensor_node, while other not sensing devices connected to the network must inherit directly from the smart_node_base class.

| API Function | Description |
|---|---|
| initialize | Configuration of the node. Initial event (if any) added to the event queue |
| handle_event | Called when an event is collected. Switch among the tasks corresponding to each event |
| send_message | Sends a transaction to the air |
| register_actuator | Registers the implemented actuator in the node class |
| register_sensor | Registers the implemented sensor in the node class at an specific address |
| get_last_sensor_value | Obtains the last sensor value available at the node |
| actuate_eup | Sends to the node the actuation directive |

Table 1: Node API Functions

In both cases, **initialize** and **handle_event** methods must be implemented. If an internal event, such as a periodic event, is required for the current device, the initial events have to be placed at the event queue in the initialize method.

In the case of a sensor node, the actuator and the sensors must be implemented and registered (through **register_sensor** and **register_actuator**).

**Functionality**

Wireless communication functionality is limited to sending and receiving messages. To send a message, the method provided is **send_message**, whose only argument is the transaction containing the message to send. Messages are received through the event queue, in the **handle_event** method, and can be identified by the event kind RCVD_MSG.

Events can be either configured or identified by the getter and setter functions **set_kind** and **get_kind**, respectively. An event can be added to the event queue by calling the method **notify**, with the event as an argument. A second parameter with the time can be added in case of timed events.

Measured values by the sensors are accessed through the method **get_last_sensor_value**.

Finally, when the energy cost is available and actuation on a specific node is required, it can be achieved by calling the method **actuate_eup**.

# 5 Example

To illustrate how to use the SmartCoDe simulation framework and to provide a base for the work of tasks 1.4 and 1.5, an example network consisting of several EuPs (class VSTSVC) and an energy management unit, has been modelled (Figure 7). In the decentralised approach described in D-1.1.1, Section 5, the energy management unit issues energy management directives in the form of cost functions to the VSTSVC nodes. The cost function is computed out of several parameters, e.g grid tariff and renewable energy availability, and has the general goal to steer the EuPs in the network such that a certain network load profile can be met.

Since the typical SmartCoDe application scenario contains several VSTSVC devices (like refrigerator, freezer, electrical heating, air-conditioning), and because the VSTSVC class is the most complex to handle, the example focuses on this class and features a network exclusively comprised to the VSTSVC devices. Probably the most interesting challenge of the decentralised approach is achieving load balancing of such VSTSVC devices (see D-1.1.1, Section 5.1).

The VSTSVC-network example allows exploring this problem once there are approaches developed in task 1.1 available.

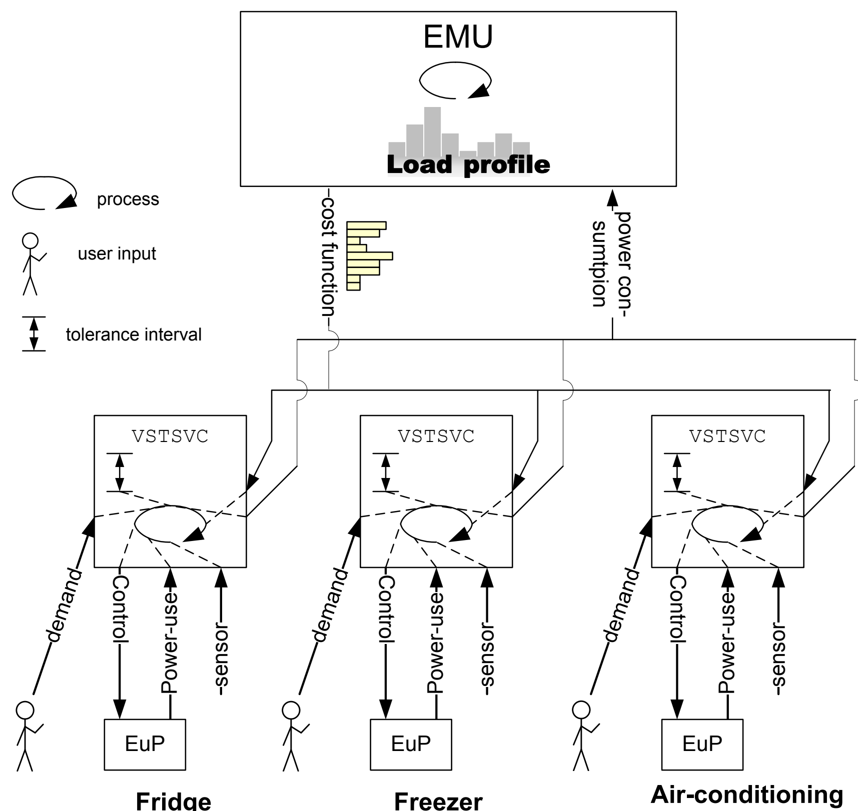



Figure 7: simulation scenario with different VSTSVC EuPs

Another question is in what form the cost function should be distributed over the network. There are many possible scenarios:

- A **general cost function** for the whole network. This has the disadvantage that it makes load balancing difficult. A possible solution are load balancing subnets, e.g. subsets of VSTSVC devices which exchange messages for load balancing.
- **Different cost functions** for different subsets of the EuPs in the network, e.g. for different device classes or different predefined balance groups up to single devices.
- The **time span covered** by the cost function could either be uniform (e.g. 24 hours), or might depend on the respective group in an approach using different cost functions; e.g. we might not need the energy cost in 12 hours to manage a fridge.

- The **time resolution** of the cost function can also be uniform (e.g. 10 minutes as it is suggested by D-1.1.1, see Section 4.1 there), but it could also get more coarse the more into the future the cost values are, e.g. hourly. This would reflect that VSTSVC nodes managing more inert thermal loads (like air-conditioning), where knowing the cost in 24 hours could be useful, will not need this information with a ten minute resolution.

Regarding the cost function, several algorithms have to be developed:

- An algorithm for the energy management unit which computes (one ore more) cost functions out of the network load profile and other available data like the wind turbine power production forecast. This has not be considered yet and will be worked on in Task 1.4.
- An algorithm *for each* EuP class as defined in D-1.1.1. which runs on the respective SmartCoDe node. It has to incorporate the cost function into its control task.

With respect to the node algorithm, the cost function might be of little significance for certain EuP classes. The VARSVC class, for example, offers little possibilities regarding energy management as it has been pointed out in D-1.1.1, Section 2.2.3. This is not the case for the shiftable loads of VSTSVC devices.

Figure 8 shows a trace of a simulation of a VSTSVC model configured as a heater which has to keep the room temperature between 20°C and 25°C. After 10000 sec, the first cost function is received; the values of the cost function which was valid at that time is shown at the bottom. In this experiment the cost function had a resolution of 4 bit and was randomly generated. Before the first cost function arrives, the node performs a simple 2-point control. After a cost function is received, the control algorithm implemented in this example takes also the medium cost of the next 30 minutes into account. When this cost is low, the heater will be switched on even if the lower threshold of 20°C s not reached yet, and vice versa.

In Figure 8 it can be seen how the temperature (upper graph) sinks during expensive times since the heater (middle graph) stays turned off, and how it rises during cheap times. It also shows how the switching frequency rises as soon as the cost function-based control kicks in. This might be an unwanted effect; as described in D-1.1.1, Section 2.2.5, some EuPs might have limits to their switching frequency.

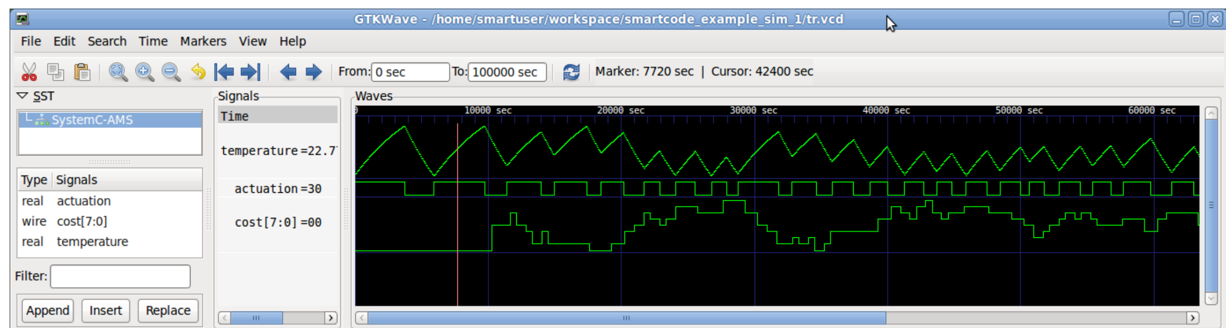



Figure 8: Simulation screenshot

This example illustrates how the simulation environment and the executable specification of a VSTSVC SmartCoDe can be used to evaluate different algorithms for cost function-based energy management. The algorithm used in this experiment was very simple. More advanced algorithms can be developed if a forecast of the temperature depending on different control decisions is incorporated. Using the fact that the temperature can be modelled using a lowpass in the time domain [7], an approach using a discrete-time model for a lowpass has been implemented. A lowpass in the discrete time domain can be modelled as

$$y_i = \alpha x_i + (1 - \alpha) y_{i-1} \tag{1}$$

where $x_i$ and $y_i$ are the input and the output of the lowpass at time $i$, respectively, and $y_{i-1}$ is the output of the lowpass at the previous time $i - 1$. The factor $1 \leq \alpha \leq 0$ is the so-called smoothing factor and

characterises the lowpass. A good estimation of $\alpha$ can be used to predict the temperature. If we solve equation 1 for $\alpha$ we get

$$\frac{y_i - y_{i-1}}{x_i - y_{i-1}} \tag{2}$$

That is, we can determine $\alpha$ out of the input to the system and successive measurements of the temperature. This approach has been tested in the simulation environment, and stable estimates of $\alpha$ could be generated that were used for temperature predictions. However, no algorithms for the VSTSVC node have been developed yet which can make use of temperature predictions; this will be part of the upcoming task 1.4. With the simulation environment developed for D-2.2, these and other questions within the tasks 1.4 and 1.5 can be investigated.

# 6 Conclusion and future work

This report documents the executable specification of the SmartCoDe node. The modelling efforts are driven mainly by considering what the main problems are to tackle next within SmartCoDe. Since the energy management algorithms in a decentralised setting are to be developed next, a simulation framework is provided where these algorithms can be tested and different approaches can be compared. This means to concentrate on network aspects, provide only a functional model for the nodes, and also model physical effects like the change of the room temperature over time. The simulation framework provides a robust basis for tasks 1.4 and 1.5 – development of the SmartCoDe distributed energy management algorithm.

Since SystemC is very flexible, the simulation framework can be extended and refined where necessary for future endeavours. For example, the architecture model of the SmartCoDe node (D-2.3) can be integrated into the simulation framework as well. It is also possible to integrate an Instruction Set Simulator (ISS) into the node model for software development. A SmartCoDe network consisting of nodes modelled in different abstraction levels (functional, architectural, ISS) can be simulated too (mixed level simulation). That way, simulation performance can be maintained while still being able to simulate a detailed node model within the network.

# References

[1] J. Aynsley. Osci tlm-2.0 language reference manual. Technical report, Open SystemC Initiative, 2009.
[2] M. Damm, J. Moreno, J. Haase, and C. Grimm. Using transaction level modeling techniques for wireless sensor network simulation. In *Proceedings of the DATE 2010*, Dresden, Germany, 2010.
[3] A. de Almeida and P. Fonseca. Residential monitoring to decrease energy use and carbon emissions in europe. In *Conference proceedings eceee Summer Studies*, La Colle sur Loup, France, 2007.
[4] J. Glaser, D. Weber, S. A. Madani, and S. Mahlknecht. Power aware simulation framework for wireless sensor networks and nodes. *EURASIP J. Embedded Syst.*, 2008:3:1–3:16, January 2008.
[5] J. Haase, M. Damm, J. Glaser, J. Moreno, and C. Grimm. Systemc-based power simulation of wireless sensor networks. In *Proceedings of the Forum of Design Languages*, 2009.
[6] IEEE Computer Society. IEEE 802.15.4: Wireless MAC and PHY Specification for Low-Rate WPANs. Technical report, IEEE Computer Society, 2006.
[7] F. Kupzog and C. Roesener. A closer look on load management. In *Proceedings of the 5th IEEE International Conference on Industrial Informatics, 2007*, pages 1151–1156. IEEE Computer Society, IEEE Computer Society, 2007.
[8] Open SystemC Initiative. *OSCI TLM2.0*. http://www.systemc.org.
[9] Open SystemC Initiative. *SystemC*™. http://www.systemc.org.
[10] Open SystemC Initiative, SystemC AMS working group. *SystemC AMS*. http://www.systemc-ams.org.
[11] PAWiS. PAWiS Simulation Framework, 2009. http://pawis.sourceforge.net/.
[12] A. Varga. *OMNeT++ Discrete Event Simulation Sytem User Manual*, 29. Mar. 2005.
[13] ZigBee Alliance. Zigbee specification revision r17, document 053474r17. Technical report, ZigBee Alliance, 2008.

## Abbreviations and Definitions

| | |
|---|---|
| API | Application programming interface |
| EMU | Energy Management Unit |
| EuP | Energy Using Product |
| OS | Operating System |
| PAWiS | Power Aware Wireless Sensors |
| SNOPS | Sensor Network Optimization by Power Simulation |
| TLM | Transaction Level Modelling |
| TUV | Vienna University of Technology |
| VARSVC | variable service |
| VSTSVC | virtual storable service |