Noname manuscript No.
(will be inserted by the editor)

A4WSN: An Architecture-driven Modelling Platform for Analysing and Developing WSNs

Ivano Malavolta · Leonardo Mostarda · Henry Muccini · Enver Ever · Krishna Doddapaneni · Orhan Gemikonakli

Received: date / Accepted: date

Abstract This paper proposes A4WSN, an architecture-driven modelling platform for the development and the analysis of Wireless Sensor Networks (WSNs).

A WSN consists of spatially distributed sensor nodes that cooperate in order to accomplish a specific task. Sensor nodes are cheap, small, and battery-powered devices with limited processing capabilities and memory. WSNs are mostly developed directly on the top of the operating system. They are tied to the hardware configuration of the sensor nodes and their design and implementation can require cooperation with a myriad of system stakeholders with different backgrounds.

WSNs peculiarities and current development practices bring a number of challenges, ranging from hardware-software coupling, limited reuse, late WSNs quality property assessment. As a way to overcome a number of existing limitations, this study presents a multi-view modelling approach that supports the development and analysis of WSNs. The framework uses different models to describe the software architecture, hardware configuration, and physical deployment of a WSN. A4WSN

Ivano Malavolta
Gran Sasso Science Institute, L'Aquila, Italy
E-mail: ivano.malavolta@gssi.infn.it

Leonardo Mostarda
Computer Science Department, University of Camerino, Camerino, Italy
E-mail: leonardo.mostarda@unicam.it

Henry Muccini
Computer Science and Mathematics, University of L'Aquila, Italy
E-mail: henry.muccini@di.univaq.it

Enver Ever
Computer Engineering, Middle East Technical University, Northern Cyprus, Turkey
E-mail: eever@metu.edu.tr

Krishna Doddapaneni
Computer Communications Engineering, Middlesex University, UK E-mail: k.doddapaneni@mdx.ac.uk

Orhan Gemikonakli
Computer Communications Engineering, Middlesex University, UK E-mail: o.gemikonakli@mdx.ac.uk

allows engineers to perform analysis and code generation in earlier stages of the WSN development life cycle. The A4WSN platform can be extended with third-party plugins realizing additional analysis or code generation engines.

We provide evidence of the applicability of the proposed platform by developing PlaceLife, an A4WSN plugin for estimating the WSN life time by taking various physical obstacles in the WSN deployment environment into account. In turn, PlaceLife has been applied on a real-world case study in the health-care domain as a running example. A case study based on home automation is presented as well.

Keywords MDE · Software Engineering · Software Architecture · Wireless Sensor Networks

1 Introduction

A recent study predicted that in 2020 there will be 50 billion devices connected to the Internet (seven times the worlds population) [20]. These devices are not only smartphone and tablets, but also *things* which are able to perform various operations, such as sensing data, actuating on the external environment, and so on. With this perspective, *Wireless Sensor Networks* (WSNs) are getting mainstream in a wide variety of applications and systems; possible applications include environment monitoring, energy metering, smart cities, health care and intelligent houses [55].

Wireless sensor networks are composed of low-data rate, low-cost and battery-operated wireless components called *sensor nodes*. A sensor node is a small digital device with communication, sensing, and limited processing capabilities. WSNs can range from small scale networks such as body sensor networks with few nodes [69], to large scale networks such as smart city applications with thousands of sensor nodes [23].

Despite the increasing usage of WSNs in modern applications, their development is still plagued by the following issues: (i) development is still performed directly on the top of the operating systems and relies on individuals hard-earned programming skills across all levels of the communication stack (e.g., application, routing, data link levels, etc.) [43]; (ii) WSN engineers must address challenging extra-functional requirements such as performance, security, energy consumption, with poor support for early testing, debugging, and simulation of WSNs in an integrated fashion [30]; (iii) in order to achieve the demanded high level of efficiency, the software of a WSN application is tied to specific hardware platforms, thus hampering the reuse of source code and software components across different projects or organizations [43]; (iv) due to the intrinsic multidisciplinary nature of the WSN problem space, WSN engineers must continuously collaborate with a high number of system stakeholders (e.g., WSN users, application domain experts, hardware designers, and software developers) with different background and training [55].

As a possible solution to the above mentioned issues, the WSN community is becoming aware of the need of using software engineering approaches in order to support the design, analysis, simulation and implementation of WSNs [68,51].

This research contributes with a novel modelling and analysis platform to support an architecture-driven development of WSNs. The platform is called A4WSN[1] and is based on a multi-view architectural approach [33] based on three modelling languages to describe a WSN from different viewpoints: (i) software components and their interactions, (ii) the low-level and hardware specification of sensor nodes, and (iii) the physical environment where sensor nodes are deployed, separately. Model-driven engineering (MDE) techniques and tools are used to realise the modelling framework through metamodelling, model weaving and model transformation. The modelling framework is supported by a programming framework that enables the implementation of analysis and code generation plugins by third party developers; they can be employed to assess and analyse the architectural design decisions and used to generate executable code, respectively.

The whole A4WSN platform is realised by exploiting advanced Model-driven Engineering (MDE) techniques, such as metamodelling, model weaving and model transformation. MDE allows us to define the modelling languages of A4WSN in a seamless and well-disciplined manner, and to realise the A4WSN programming framework so that it supports extensibility and customisation by design. We provide evidence on the applicability of the proposed modelling approach and on the extensibility of its programming framework by developing a A4WSN plugin called PlaceLife. Placelife analyses A4WSN models of a WSN and automatically assesses the life time of the network.

PlaceLife uses the A4WSN physical environment model that includes physical objects and material, thus providing an accurate WSN lifetime estimation. The *PlaceLife* plugin has been applied onto a realistic case study in the home automation application domain. The scenarios that can be considered using A4WSN are more realistic compared to existing simulation tools for WSNs, since the existing tools consider simplified models of the environment (e.g., a free space model) due to the limitations mentioned above. In a previous work [19] we made the first exploration for the feasibility of the modelling and analysis platform, with special emphasis on energy consumption analysis. In that work, the used modelling languages and the programming framework were in an incubation phase. The main contributions of this paper can be summarised as follows:

- a multi-view modelling platform for engineering WSNs is presented in details (including a specification of the software architecture, low-level and hardware specification, and physical environment);
- a programming framework is provided which enables the development of plugins realising new code generation and analysis engines;
- the A4WSN prototype tool[2] realising the approach presented has been fully implemented; it is based on the Eclipse platform and can be integrated with other MDE technologies already available in the Eclipse community;
- the *PlaceLife* plugin for the prediction of the life time of a WSN is presented, where the physical environment (together with other factors) is also taken into account unlike the existing studies.

[1] It stands for **A**rchitecting platform for (**4**) **W**ireless **S**ensor **N**etworks
[2] A4WSN prototype: http://a4wsn.di.univaq.it

The rest of this paper is organised as follows. Section 2 provides background information on WSNs, and the motivational example used throughout the paper. Section 3 provides an overview of the A4WSN platform. Section 4 describes the proposed modelling languages for WSNs, while Section 5 presents the programming framework. Section 6 presents the technologies we use to implement the A4WSN platform. Section 7 focusses on the PlaceLife analysis plugin. Finally, Section 8 presents related work, while the paper concludes in Section 9.

2 Background and Motivational Example

This section provides a background on wireless sensor networks (Section 2.1), considerations to motivate the need of our approach (Section 2.2), and introduces the case study used throughout the paper (Section 2.3).

2.1 Background on Wireless Sensor Networks

Wireless Sensor Networks (WSN) consist of spatially distributed sensors that cooperate to accomplish some tasks. Sensors are small battery-powered devices with limited processing capabilities and memory. In the current state of the art, WSN nodes processor frequencies range from 4Mhz to 32 Mhz, whereas typical amounts of memory range from 2 Kb to 512 Kb [43]. They can be easily deployed to monitor different environmental parameters such as temperature, movement, sound and pollution. Sensors can be distributed on roads, vehicles, hospitals, buildings, people and enable different applications such as medical services, battlefield operations, crisis response, disaster relief and environmental monitoring.

WSNs can be of the type *event-driven* or *continuous*. Event-driven WSNs report data to the base station only when certain events occur. For instance, intrusion and fire detection are examples of applications that are implemented by using event-driven WSNs. Continuous WSNs report data to the base station at regular intervals. For instance patient monitoring and temperature control are examples of applications that are implemented by using continuous WSNs. In its most common configuration, sensors communicate the sensed information to the base station BS. This can be achieved by using single-hop or multi-hop topology. When single-hop topology is considered, the nodes communicate with the BS directly whereas in multi-hop, some of the nodes may act as intermediate routers which are responsible for forwarding the information from other nodes besides the usual sensing responsibility.

The unique characteristics of WSNs introduce new challenges [63] in different fields such as programming, security and software engineering. Researchers need to face limited sensor resources in terms of computation capabilities and memory as well as the limited life time of the sensors.

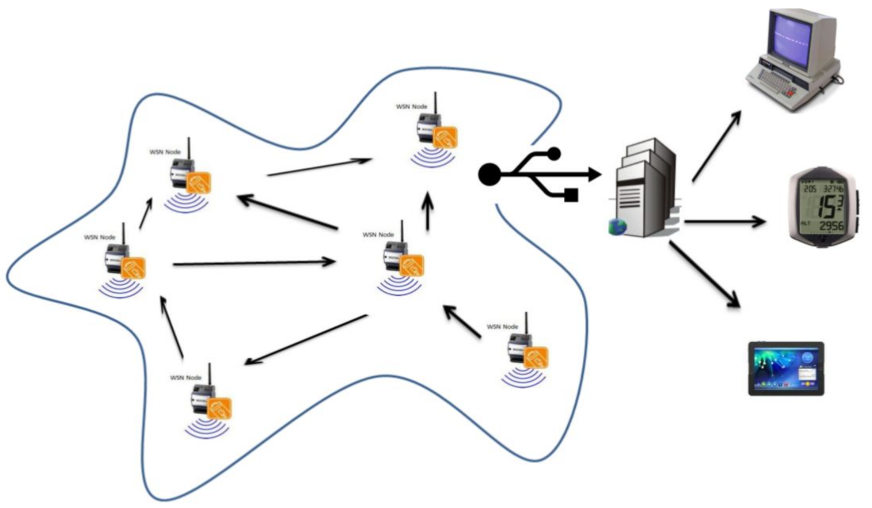

Fig. 1 Typical Architecture of a WSN Application

2.2 The WSN challenges and A4WSN solutions

The issues outlined in Section 1 bring a number of WSN design and development challenges. In the following some of the most relevant challenges which can be solved by abstraction and modelling are described. A summary on how the A4WSN framework addresses them is also presented in Section 3.

Abstracting implementation details into a design view: the development of a WSN application requires skills across all levels of the communication stack. A WSN application developer has to be knowledgable on software programming as well as all the layers of the ISO/OSI reference model that is from physical to application layers. Beside the need of programming abstraction, it is well-accepted the need of abstracting an implementation view into an architectural design. As remarked in [51], *"end users require high-level abstractions that simplify the configuration of the WSN at large, possibly allowing one to define its software architecture based on pre-canned components"*. Abstraction is fundamental for future WSN development, since sensors and WSNs in general are becoming important components in pervasive computing, and mobile systems, with new types of stakeholders (e.g., mobile systems engineers, developers) and reduced domain-specific technical skills. Under this perspective, approaches for abstracting the implementation details from the underlying hardware and physical infrastructure are strongly advised [43,10]. However, when current practices on WSNs are considered, it is quite evident the lack of engineering methods and techniques to manage these challenges. Some initial effort has been conducted for architecting WSNs [37,28], and this paper goes along that line providing more advanced solutions. A thorough comparison with related work is provided in Section 8.

Increase reuse: state of the art approaches mostly mix together software, hardware, and networking perspectives during the coding or design phase. Hardware and software components are locked and tied down to specific type of nodes, thus hampering the reuse of source code and software components across different projects or organizations [43].

Early WSNs quality property assessment: in traditional implementation-specific approaches engineers might afford to take structural and behavioural decisions at deployment time. However, in WSN development it is important to take those sensible decisions as early as possible, enabling the earlier, predictive, analysis of both functional and extra functional properties. This possibility becomes especially valuable in all those cases where the sensor nodes cannot be easily accessed once deployed (e.g., WSN nodes embedded into concrete walls or WSNs deployed in hostile environments).

In order to address the aforementioned challenges we developed A4WSN, an architecture modelling platform that uses different models for specifying WSNs. It provides a design view of the system, and hides low-level details and complexities. A4WSN, being a multi-view approach including different models which cover different concerns, increases **separation of concerns** favouring the possibility to **reuse** software and hardware components across projects and organisations. A4WSN enables **model-based analysis** techniques and favour the earlier, predictive, analysis of both functional and extra functional properties.

2.3 The health care system case study

In this paper we will use the health case system case study as running example in order to help the reader in promptly understanding the main concepts and design decisions we took when engineering the A4WSN modelling languages. A case study based on home automation is also presented in order to show the effectiveness of the architectural approach and the PlaceLife plugin.

Recent technological advancements in WSNs have opened up new prospects for a variety of applications, including healthcare systems [36,5,62]. WSN implementations on pervasive computing based health care systems avoid various limitations and drawbacks associated with the wired sensors providing a better-quality of care, quicker diagnosis, more intense collection of information (can be employed for statistical analysis) and at the same time keeping the cost and resource utilisation to minimal. Monitoring facilities introduced by using WSNs are particularly useful for early detection and diagnosis of emergency conditions, as well as keeping track of the diseases for patients. WSN based health care systems are also useful for providing a variety of health related services for people with various degrees of cognitive and physical disabilities [4].

In the context described above, our case study (graphically illustrated in Figure 2) represents the concept of an in-hospital WSN that allows the personnel of the hospital to monitor patients' vital sign data with the help of pulse-oximeters. Our

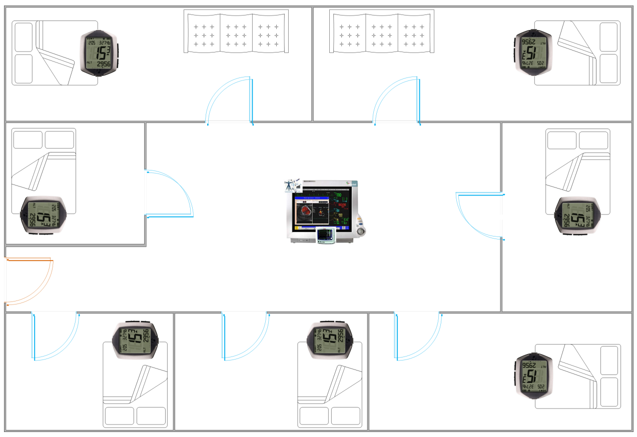

Fig. 2 Hospital scenario considered: i) the central component represents the monitoring station, ii) a pulse-oximeter is included in each room around the central one.

monitoring system consists of two types of nodes: a monitoring station and seven oximeter nodes, forming a star-network. Each pulse-oximeter monitors the patient continuously and a measurement is sent to the monitoring station every three seconds. In case the oximeter reads a value other than a defined threshold, an alert message is sent to the monitoring system, and the system goes into a *warning* mode in which sensor readings are sent to the monitoring station more frequently (i.e., once every 200 milliseconds), hence facilitating continuous monitoring of patients and allowing real-time responses in case of emergency conditions.

3 Overview of the Platform

In this section we provide an overview of the A4WSN platform. The main goal of our research is to take advantage of Model-Driven Engineering (MDE) techniques to support an architecture-driven development and analysis of wireless sensor networks. As shown in Figure 3, the A4WSN platform is composed of two main parts: a modelling environment for describing the architecture of a WSN and a programming framework.

The **modelling environment** exposes three modelling languages for describing specific architectural views of a wireless sensor network: the *Software Architecture Modelling Language for WSN* (*SAML*), the *Node Modeling Language (NODEML)*, and the *Environment Modelling Language (ENVML)*. In the following we will describe the rationale that drove us to propose each modelling language. The SAML language focuses on the *application layer* of the WSN. It is used to break down the application into a set of software entities (e.g., components), to show how they relate to each other, to better reason on their distribution throughout the network, and to reason on the business logic of the WSN. The NODEML language concerns all the low-level aspects underneath the application layer of the WSN. In this context, stakeholders reason about routing protocols, middleware, hardware configuration of

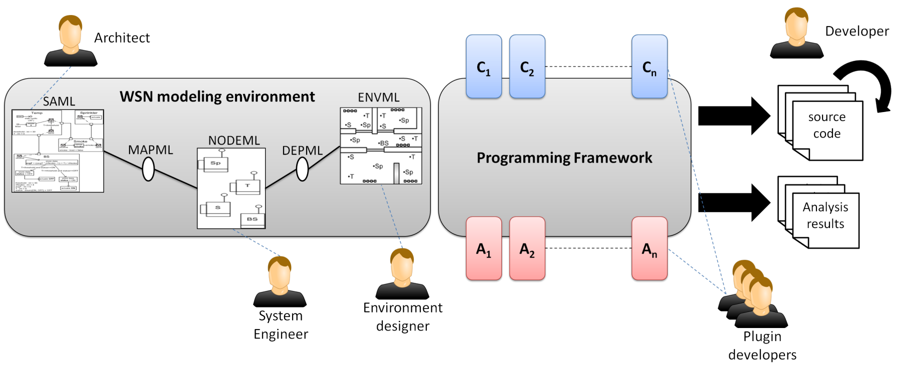

Fig. 3 Overview of the A4WSN platform

the nodes, etc. The ENVML language is about the site in the real world where the WSN will be deployed. This viewpoint could be specially useful for developers and system engineers when they have to reason about the network topology, the presence of possible physical obstacles (e.g., walls, trees) within the network deployment area, and so on.

In order to provide a useful instrument for modelling WSNs, we identified the system concerns which are held by engineers and developers while designing a WSN. This set of concerns is based on our experience in the field of WSN development and on some informal investigation we performed on how expert engineers and developers have worked on previous WSN projects. The resulting set of system concerns in the domain of WSN can be summarised as:

– *Energy Consumption* concerns how much power is consumed by the application running on the nodes with respect to the used batteries or harvested energy sources. It is a crucial factor that constrains the networks life time for the WSN mission [19].
– *Dependability* is a generic attribute to describe “the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers”. Dependability includes attributes such as reliability, availability, safety, security as special cases[3].
– *Coverage* represents how well an area is monitored or tracked by sensors [29].
– *Networking and Communication* represent various networking aspects which may affect the WSN. Example of the issues which are related to this specific stakeholder comprise: routing protocols, MAC protocols, usage of a specific communication middleware, connectivity of the network, etc.
– *Performance* deals with all performance-related aspects of the WSN, such as throughput, latency, response times, nodes utilisation, etc.
– *Data Collection* concerns data structures and how data is collected, managed and transformed by the various nodes within the WSN.

[3] http://www.dependability.org/

We designed the proposed modelling languages so that they can frame all the above listed concerns. The three proposed modelling languages are linked together via two auxiliary modelling languages in order to create a combined software, nodes, and environmental view of a WSN. These languages are called Mapping Modelling Language (*MAPML*) and Deployment Modelling Language (*DEPML*), and they link together SAML to NODEML and NODEML to ENVML, respectively. More specifically, MAPML links are used for assigning software components to the corresponding hardware node configuration they will be executed on. Whereas, DEPML links are used for virtually deploying WSN nodes into specific areas within the physical environment. We decided to use those two auxiliary modelling languages for a clear *separation of concerns* and duties while architecting the WSN (e.g., a software architect can focus on the application layer in the SAML model only, while a system engineer may focus on the nodes configurations in the NODEML model) and making the used models *reusable* across projects and organizations (for example, the same nodes configuration defined in a NODEML model can be shared across different applications produced by a software company). The main concepts of each modelling language will be described in Section 4.

The **programming framework** provides a set of facilities for supporting the development and integration of either *code generation* or *analysis* engines. Each engine is realised as a plugin of this framework. Our proposed programming framework knows at run-time which plugins are installed into the framework, and automatically provides to the user the available target implementation languages or the available analysis techniques.

Code generation and analysis plugins are structurally similar. An analysis plugin manages the analysis of WSNs (e.g., coverage, connectivity, energy consumption analysis), instead of a code generation plugin which is tailored to the generation of implementation code conforming to a set of specific target languages. More specifically, in A4WSN the main difference between code generation and analysis plugins resides in their returned output: the main output of a code generation engine can either be a set of source files, or binary packages, whereas the main output of an analysis engine can be a violated property, a counter-example, a set of numerical values, and so on. The detailed description of the programming framework is presented in Section 5.

The A4WSN platform is generic since it is independent from the programming language, hardware and network topology. Starting from a set of models (each one reflecting a certain WSN viewpoint), the code generation and analysis components can be plugged into the framework for generating executable code or analysing outcomes.

4 The Modelling Environment

As shown in the previous section, our modelling environment is composed of three main languages, which are SAML, NODEML and ENVML. Each language allows the user to frame the problem of describing the architecture of a WSN from a specific viewpoint [33].

More specifically, the **SAML** modelling language focuses on the application layer of the WSN. It is used to break the application down into a set of software entities (e.g., components), to show how they relate to each other, to better reason on their distribution throughout the network, and to reason on the business logic of the WSN. The **NODEML** modelling language concerns the low-level details underneath the application layer of the WSN. In this context, stakeholders reason about routing protocols, middleware, hardware configuration of the nodes, etc. The **ENVML** modelling language is about the physical environment where the WSN will be deployed. This viewpoint could be specially useful when they have to reason about the network topology, the existence of possible physical obstacles (e.g., walls, trees) within the network deployment area, and so on. The **MAPML** modelling language weaves together an SAML model and a NODEML model. It allows designers to define a set of mapping links, each of them weaving together components in the SAML model and node definitions in the NODEML model. The **DEPML** modelling language weaves a NODEML model to an ENVML model. It allows designers to consider each node type defined in the NODEML model and to *instantiate* it in a specific area within the physical environment defined in an ENVML model. Each node type can be instantiated "n" times within a specific area with a certain distribution strategy.

It is important to point out that the modelling framework has been realised by: carefully and extensively checking the state of the art in WSN development, discussing with WSN and embedded systems engineers, with many iterations of changes and carefully and extensively checking the state of the art in WSN modelling, as shown in the Section 8. In the next sections we will go into the details of each proposed modelling language. Each subsection will first introduce and explain the modelling language metamodel, followed by an application to the health care case study.

4.1 Software Architecture Modelling Language (SAML)

The SAML modelling language allows architects to define the software architecture of the WSN application. We formalise the structure of the SAML language and its constructs by defining its underlying metamodel. Figure 4 and Figure 5 show the parts of SAML metamodel related to its structural and behavioural concepts, respectively[4].

The **software architecture** of a WSN is defined as a collection of software components and connections. It represents the application layer of the WSN and it is the root element of every SAML model. A **component** is a unit of computation with internal state and well-defined interface [11]. The internal state of a component is represented by the values of its application data and its current behavioural mode. An **application data** can be seen as a local variable declared in the scope of the component; application data are manipulated by actions, events, and conditions defined in the behaviour of the component, such as **StoreData**, **ReceiveMessage**, etc. Application data can be either primitive or structured. **Primitive data** represents a data declaration which may have only a primitive type. The primitive types available in

[4] The name of abstract classes is shown in italics

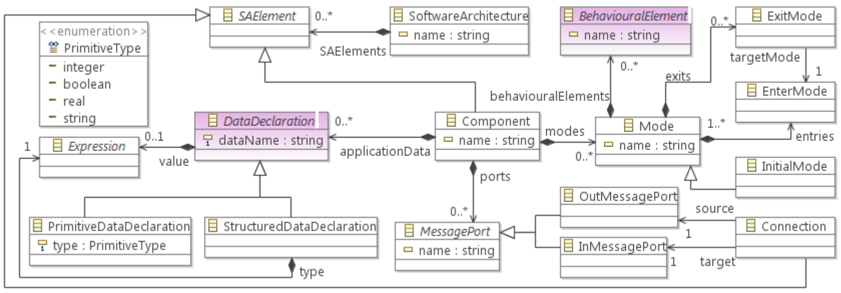

Fig. 4 SAML Metamodel: structural concepts (external metaclasses in pink)

the SAML language are: integer, boolean, real, string, and byte array. We do not consider other primitive types (such as double, short, array, etc.) because we want to keep the SAML language as simple as possible, avoiding to overwhelm architects with a large number of data types, in which many of them could likely be used only in very specific situations. **Structured data** represents a data declaration which may have a type composed of other (either primitive or structured) types. A structured data type can be either an enumeration, a structure, an array, or a map[5]. An **enumeration** data type contains a sets of identifiers that represent the possible values of the enumeration. A **structure** data type can contain an arbitrary set of primitive data types; the set of internal primitive data types of an SAML structure can be defined only when it is declared. An **array** data type contains a collection of elements of the same type; each element can be accessed by its index within the collection. A **map** data type contains a collection of key-value elements; keys are represented by a string, whereas values can be either primitive or structured; each value can be accessed by its associated key. A more detailed description of application data, together with the concrete syntax for defining and accessing them, are provided in [38].

A **mode** represents a specific status of the component. Examples of modes can be, sleeping mode, energy saving mode, etc. Modes are defined at the application layer (in the SAML model) and, while they can be directly related to energy-related modes of a sensor node, they can also be used the represent any logical state of a sensor. At any given time, one and only one mode can be active in a component. The component reacts only to those events which are defined within its currently active mode. An initial mode is the first mode which is active when the component starts up. Mode transitions occur by passing from a special kind of action called **ExitMode** to a special kind of event called **EnterMode** (the concepts of exit and enter mode will be described later in this section). In this way, actions and events can be linked to modes entry and exit points, creating a continuous behavioural flow among modes. Each mode can contain a set of **behavioural elements** that represent actions, conditions and events which together make up the control flow within the component from an abstract point of view. In a way, SAML modes may seem similar to UML state machines since they represent the states of a component, and it may switch from

[5] For the sake of simplicity, we do not show neither the metaclasses involved in the structured data declaration types, nor the metaclasses describing the expressions and operations which can be performed on data declarations in general

a mode to another via a transition; however, in order to manage the complexity of the models, SAML modes can contain only SAML behavioural elements and within a component one and only one mode can be active (no concurrency).

Components interact with other components through **message ports**; they specify the interaction points between a component and its external environment. Input message ports are used to receive incoming messages, while output message ports are used to send outgoing messages. Communication happens by message passing, which means that, a component can send messages from one of its output message ports to input ports of other components. The actual communication method of a message (i.e.,broadcast, multicast or unicast) is specified in the send message action described later in this section. In this context, a **connection** represents unidirectional communication channel between two message ports of two different components. The data contained in a message is accessible by specific actions and events defined in the behaviour of the involved components (see the **SendMessage** action and **ReceiveMessage** event in Figure 5).

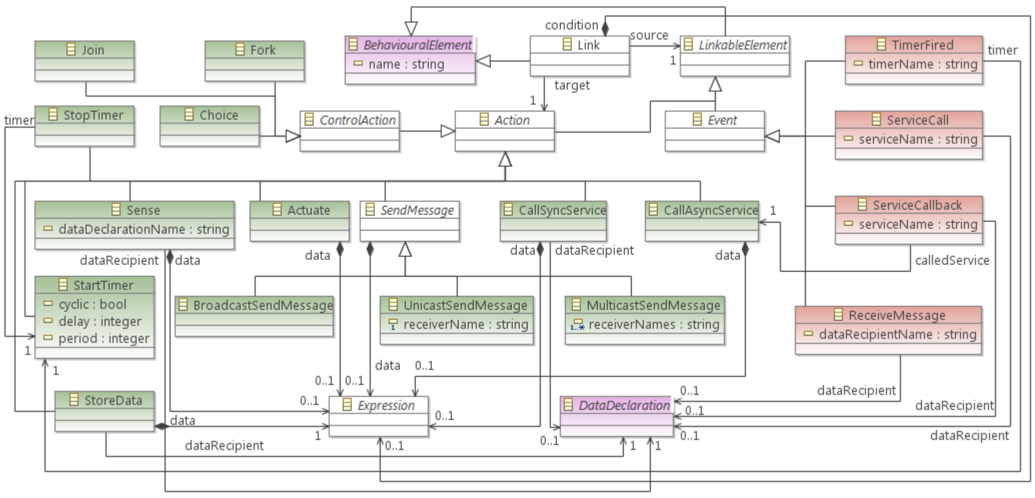

Fig. 5 SAML Metamodel: behavioural concepts (actions in green, events in red)

As previously considered, in SAML each component can contain its corresponding behavioural description. Figure 5 shows the part of SAML metamodel related to behavioural concepts. Each component can contain a description of its behaviour in terms of events, conditions and actions. **DataDeclaration** and **BehaviouralElement** correspond to the metaclasses with the same name in Figure 4.

An **action** is a special kind of behavioural element which represent an atomic task that can be performed by the component. An action can be performed in response to an event trigger, or because a previous action in the behavioural flow has been executed. Examples of action include: start or stop of a timer, send a message via a specific message port (either as unicast, multicast, broadcast, or scoped), get data from a sensor, call an external service, start a timer, and so on. It is important to describe a new kind of action we introduced called *scoped send message*; basically, this action tells that the set of nodes receiving the message is computed at run-time, depending on the value of a boolean expression; only the nodes whose application data values satisfy the boolean expression will receive the specific message, thus enabling dynamic scope-based interactions within the WSN [42]. For example, a scoped send message may be used in order to send a message to all the nodes whose *floorName* application data is equal to "*ground*" and whose *temperature* application data is greater than 21 degrees. Special kinds of actions manage the control flow within the behavioural description of each component: the *fork* action splits the control flow into multiple parallel executions, the *join* action merges (previously split) control flows, and the *choice* action specifies a branch in the control flow after which one and only one outgoing control flow will be executed. A more detailed description of the types of action supported by SAML is provided in [38].

An **event** is an SAML behavioural element representing an occurrence that can happen during the execution of a component. An event is triggered in response to either an external stimulus of the component (e.g., the message reception on a input message port), or some internal mechanism of the component (e.g., a timer fired). Examples of event include: entering a specific mode, receiving a message at a given port, an activation of a timer, the receiving of a call from an external service, the receiving of an interrupt from either a sensor or an actuator, etc. A more detailed description of the types of event supported by SAML is provided in the technical report that is accessible on [38].

Events and actions are connected via **links**, they represent the control flow among events and actions. A link helps architects in defining the order in which actions can be executed and the actions that must be executed when an event is triggered. Thus, a link can exist either from an event *e* to an action *a* (in this case, *a* is executed only after *e* has been triggered), or from an action *a* to another action *b* (in this case, *b* can be executed immediately after *a* has been executed). Optionally, a condition can be specified in a link; the behavioural flow goes through a link only if its condition evaluates to true. Conditions are defined as boolean expressions which may refer to application data declared in the component, constants, and other operations (see the **Expression** metaclass in Figure 5). SAML exposes five types of expression:

1. **Constant**: a constant representing the value of application data that is considered;
2. **DataRef**: a reference to the value of an application data declaration; it is conceptually similar to a variable reference in Java;
3. **StructureMemberRef**: a reference to a member of a structure; it is conceptually similar to the access to members of a class in Java;
4. **EnumMemberRef**: a reference to a member of an enumeration; it is conceptually similar to the access to a value of a Java enumeration;
5. **Operation**: an application of some kind of operation. Available operations are:
   - arithmetic operations: sum, subtraction, multiplication, division;
   - boolean operations: logical *and*, logical *or*, logical *not*;
   - relational operations: $>, \geq, <, \leq, =, \neq$.
   - string operations: *length*, *contains*, *substring*, *concat*, *startsWith*, *endsWith*;
   - byte array operations: *hash*, *contains*, *indexOf*, *concat*, *subtract*.

For the sake of brevity, and since the various expressions available in SAML follow classical mathematical, logical, and programming expressions, we do not graphically show them in Figure 5, and we do not go into the details of their semantics.

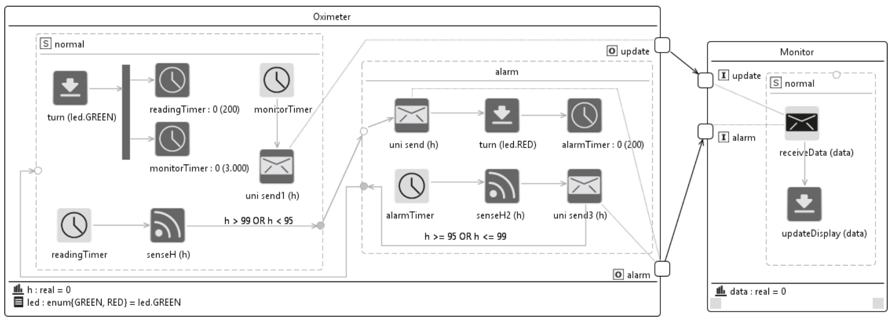

Fig. 6 Software architecture of the hospital scenario WSN

Figure 6 shows the SAML model of the WSN of the hospital scenario introduced in Section 2.3. It is important to note that this figure is actually a screenshot of the real A4WSN tool available here: http://a4wsn.di.univaq.it. From a structural point of view, the whole WSN is composed of two main components: the *Oximeter* component represents the software running on each oximeter node, while the *Monitor* component represents the software running on the monitoring station.

Oximeter stores the current percentage of oxygen in the patient's blood as a real number in the *h* application data, and the current state of its status led in the *led* application data, which can be either RED or GREEN. At startup, this component turns the led into green via the *turn(led.GREEN)* actuate action and starts two cyclic timers in parallel. Every time the *monitorTimer* is triggered (every 3000 milliseconds), the component sends the current value of the *h* application data to the Monitor component via the update message port. When the *readingTimer* is triggered (i.e., every 200 milliseconds), the component senses the current oxygen percentage in the patient's blood via the *senseH* action: if the read value is not below or above the norm (i.e., if it is not between 95% and 99%), then the component switches to the *alarm* mode. In this specific mode, the component firstly sends the current read value to the Monitor component via a dedicated *alarm* message port, then it turns the led into red, and starts a new cyclic timer with a period of 200 milliseconds. From this point onwards this component senses the percentage of oxygen in the blood of the patient and sends it to *Monitor* every 200 milliseconds. If the read value comes back in the acceptable range, then the component switches back to the *normal* mode.

The *Monitor* component is straightforward. It has a single operating mode in which every time a message from the *Oximeter* component is received, its data is shown on a display via the *updateDisplay* actuate action. This component temporarily stores the value received by the various oximeter nodes in *data*.

4.2 Node Modelling Language (NODEML)

NODEML is our language for describing the low-level details of each *type of node* that can be used within a WSN. A NODEML model contains exclusively low-level, node-specific information. Different WSN applications can reuse the same NODEML models and organize them differently, depending on the requirements of the application. Figure 7 shows an overview of the metamodel underlying our NODEML language. In the following we provide a description of the main concepts defined in the NODEML metamodel. For the sake of brevity, we do not go into the details of each element of the language, the details are presented in the the technical report that is accessible on [38].

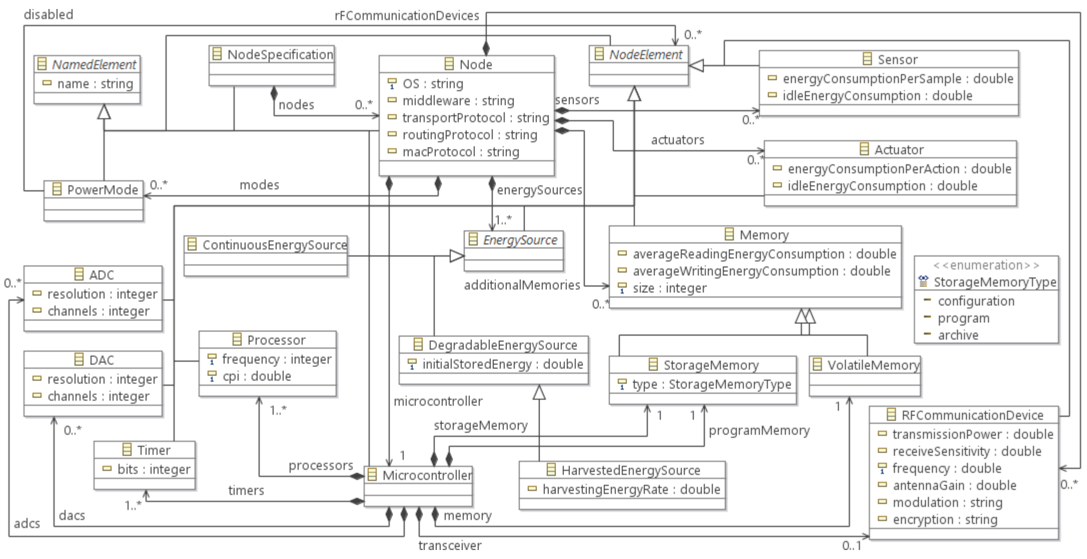

Fig. 7 NODEML Metamodel

A **node specification** is the root element of a NODEML model. It is a container of instances of the *Node* metaclass. A node represents a specific node type that can be used within the WSN. A node can have a name (inherited from the *NamedElement* metaclass), its operating system (e.g., TinyOS, Contiki, Mantis, LiteOS), *middleware* (such as TeenyLIME, MiLAN, RUNES [44]), *transportProtocol* (such as UDP and TCP), *macProtocol* (such as T-MAC, S-MAC, WiseMAC, SIFT [17]) and *routing-Protocol* (such as SPIN, LEACH, GEAR [3]) it uses to communicate with other nodes within the WSN can be specified.

From a structural point of view, according to the NODEML metamodel a WSN node is composed of a set of node elements; in this part of NODEML, we took inspiration from the abstract view of the low-level parts in a typical WSN node as described by Picco and Mottola in [43]. Figure 8 graphically shows how a WSN node can be represented in NODEML. According to NODEML, a WSN node contains one or more energy sources (e.g., batteries), a microcontroller (i.e., the component mainly devoted to computation and memory management), a set of sensors, a set of actuators, a set of additional memories representing external storage memories of the node, a set of radio communication devices to communicate with other nodes within the WSN, and a set of power modes in which the node can be at any given time.

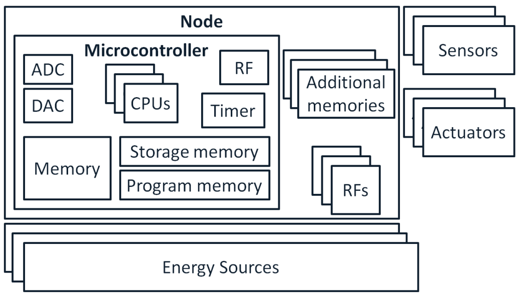

Fig. 8 an abstract view of the low-level parts of a WSN node.

An **energy source** represents any equipment or device used to provide electrical energy to the node. NODEML supports three types of energy sources, namely:

- **ContinuosEnergySource** that potentially never runs out. A classical example of this kind of energy source is AC/DC supply.
- **DegradableEnergySource** that can terminate at any time. As an example, batteries can be represented by a degradable energy source in NODEML.
- **HarvestedSource** that can terminate at any time and harvest additional energy in some way. As an example, in NODEML batteries coupled with a solar panel can be represented as an harvested energy source.

In NODEML, **sensors** are the hardware component performing the actual readings from the environment. Intuitively, a sensor can be seen as a unit that measures a physical quantity and converts it into a signal which can be analyzed and manipulated by a microcontroller. A WSN node can be equipped with zero or more sensors. Today many types of sensors exist, each of them tailored to acquire specific data from the environment; for example, a sensor can get information about the lighting conditions of the environment, its current temperature, presence of smoke or gas, the geographical position of the node, and so on [55].

An **actuator** is the hardware component that can physically operate on the environment. Conceptually, it performs the inverse operation of a sensor, i.e., sensors acquire information from the environment and allow the microcontroller to perform some computation with it, whereas actuators are triggered by some computation of the microcontroller and then perform an action in the environment. A WSN node can be equipped with zero or more actuators. Examples of actuators include: water sprinklers, lights, electronic door locks, motors, airscrews, etc.

An **RF communication device** is a radio device to communicate with other WSN nodes. For example the ChipCon 2420[6]. Technically, in NODEML an RF communication device represents an RF transceiver that can operate on specific bands, such as the 2.4 GHz ISM, XX etc., and are compliant with some IEEE standard which specifies their physical layer and media access control, such as the IEEE 802.15.4 [26]. Typically, those kind of transceivers are designed for low-power and low-voltage wireless applications. In NODEML, an RF communication device has some attributes related to the capabilities of the device (e.g., transmission power, frequency, etc.), their description is provided in the technical report that is accessible on [38].

A WSN node commonly requires **memory** as well. NODEML supports two kinds of memory:

- **Volatile memory** that represents the classical volatile memory in computer systems. When power supply is interrupted the stored memory is lost. Usually, the size of a volatile memory in a WSN node ranges from 2Kb to 512Kb [43].
- **Storage memory** that represents the storage device of the WSN. This kind of memory is usually utilised for persisting data within the WSN. Unlike the volatile memories, storage memories preserve stored data also when power supply is interrupted. Usually, the size of a storage memory in a WSN node ranges from 128Kb to several gigabytes [43]. According to NODEML, a storage memory can be of different types: *configuration*, *archive*, and *program* memory, depending on how the application uses it.

A **micro-controller** is an electronic device integrated into a single chip, it is commonly used in embedded systems. Examples of micro-controllers are: ATMega128, Texas Instruments MSP430, etc. According to NODEML, a micro-controller can contain one or more processors, zero or more ADCs (abbreviation for Analog-to-Digital Converter) , zero or more DACs (abbreviation for Digital-to-Analog Converter), one or more timers, a volatile memory, a program memory, a storage memory, and an optional radio transceiver.

The **processor** is the element which physically performs the computational logic of the node. Examples of processors include 8 bit AVR Mega, ARM920T, etc. In NODEML a processor is characterized by its *frequency* (i.e., its clock rate) in MHz, and its cycles per instruction (*CPI*) representing the number of clock cycles needed for executing a single instruction.

An **ADC** is a device for converting a continuous physical signal into a digital value that “discretizes” it. A **DAC** is a device that performs the inverse operation of an ADC; it converts digital values into continuous physical signals.

Timers are devices apt to periodically trigger the clock of the WSN node. A timer can be implemented either as a hardware or software component and usually works even when the device is in sleep mode, allowing the node to switch from sleep to active power mode. In NODEML, designers can specify the number of bits of a timer (usually they are 16 or 32 bits).

Finally, a node can specify a set of **power modes**, each of them describing a specific configuration of the elements of the node in terms of their power state. Each

[6] ChipCon 2420 data sheet: http://www.ti.com/product/cc2420

power mode identifies a set of node elements (such as memory, DAC, RF comm. device, etc.) and identifies which elements are *disabled* (all the other elements of the node are assumed to be active). For example, a given WSN node can have a *Sensing* power mode in which all the radio transceivers are disabled (thus saving all the energy needed to perform networking operations), or a WSN node can have all the sensing devices disabled, and thus focusing only on networking operations, and so on.

Figure 9 shows the NODEML model developed for our hospital scenario. Two node configurations have been defined:

– *OximeterNode* is equipped with a *IRProbe* sensor for sensing the percentage of oxygen in the patient's blood and a led actuator for showing the current status of the node to the personnel of the hospital. This node is powered by two AA batteries with up to 18720 Joules and uses a Texas Instruments ChipCon 2420 RF transceiver. The micro-controller used is the low-power Atmel AVR ATmega128 equipped with an ADC for translating the analog values read by the IRProbe sensor into their corresponding digital values. The oximeter node is always active (see the *active* power mode).
– *MonitorStation* has a single actuator device for graphically showing the values received by various oximeter nodes on a digital display. Similar to *OximeterNode*, it uses a Texas Instruments ChipCon 2420 RF transceiver and uses low-power Atmel AVR ATmega128 micro-controller. The monitoring station is always active (see the *active* power mode) and is powered by a classical electrical plug connected to the main electrical system of the hospital. Finally, it is equipped with an additional storage memory for storing a log of all the values received by the oximeter nodes over time.

Both nodes use TinyOS[7] as operating system, GEAR as routing protocol, and T-MAC as MAC protocol.

4.3 Environment Modelling Language (ENVML)

The ENVML modelling language allows the designers to specify the physical environment in which the WSN nodes are deployed. Figure 10 shows the metamodel underlying the ENVML modelling language.

The **Environment** represents the overall area in the 2D space in which the WSN nodes are deployed. The *width* and *height* attributes represent the dimensions in meters of the minimum bounding box of the environment. In geometry, a minimum bounding box is the smallest rectangle that can be drawn around a set of points such that all the points are inside it, or exactly on one of its sides. The four sides of the rectangle are always either vertical or horizontal, parallel to the x or y axis [1]. The *imagePath* attribute contains the path of an image file representing the modelled environment in more details. It can refer to a png or jpeg image exported from a CAD software (like AutoCAD [2]), or from other design tools. The rationale behind the *imagePath* attribute is that environment designers may provide a more detailed view

[7] http://www.tinyos.net/

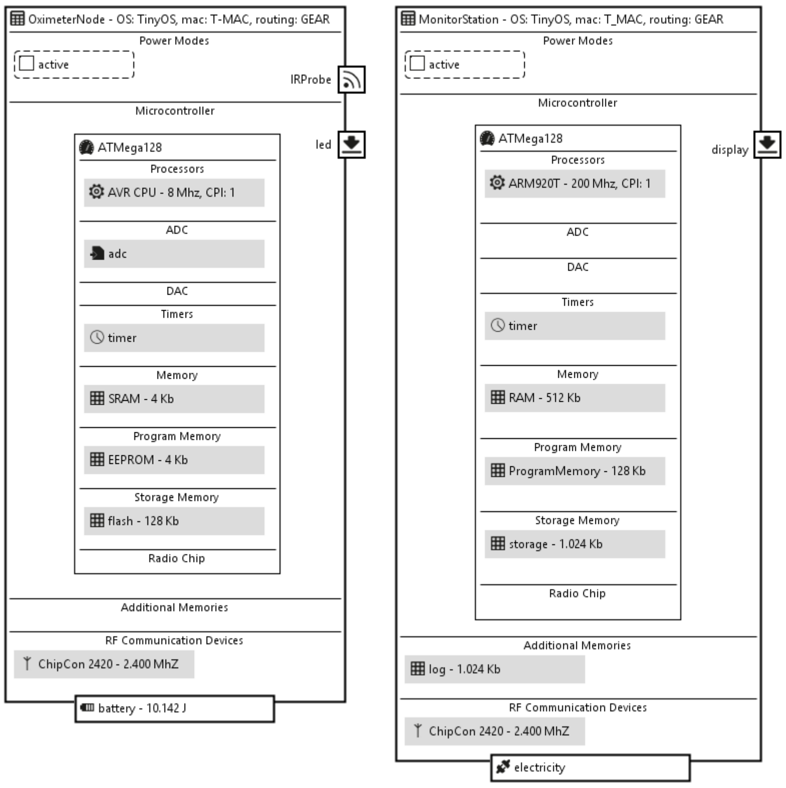

Fig. 9 Nodes configuration of the hospital scenario WSN

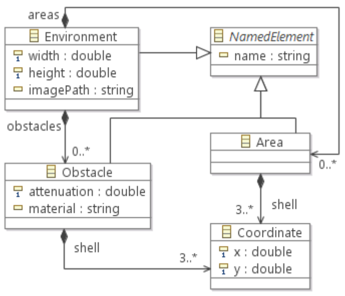

Fig. 10 ENVML Metamodel

of the environment by means of external CAD software, and then our ENVML models will be a projection of these models which focusses on obstacles and inner areas only.

Any kind of relevant **obstacle** can be placed in the environment. Each obstacle is characterized by the name of the *material* it is made of (e.g., concrete wall, wooden door, glass, etc.), and its *attenuation* coefficient. The latter attribute is a decimal number ranging from zero (when it is totally irrelevant when considering radio signal attenuation, e.g., a sheet of paper), to one (when it totally blocks radio signals, e.g., a panel made of lead). The attenuation coefficient is one of the most important parameters when considering path loss models, network connectivity and coverage, and so on. The shape of the obstacle is given by its *shell*: a sequence of **coordinates** representing the perimeter of the obstacle in the 2D space.

In ENVML an **area** identifies a portion of physical environment in which nodes of the same type can be distributed according to a distribution policy (defined in the DEPML modelling language, see Section 4.5). Similar to obstacles, the perimeter of the area is defined by means of its shell.

Figure 11 shows the ENVML model representing the physical environment of our hospital scenario. It is a rectangle with 16 and 13 meters of width and height, respectively and it contains three kinds of obstacles that are concrete walls dividing the whole environment into rooms and corridors, a main wooden door on the left, and a glass door for each patients room. Each obstacle is represented by a unique name, its attenuation coefficient and the coordinates of all the points of its perimeter. The physical environment of our hospital scenario contains two main deployment areas:

- *BSArea* is a square area at the center of the environment and will contain the central monitoring station.
- *OximetersArea* its perimeter is the same as the whole physical environment and will contain all the oximeter nodes, one for each patients' room.

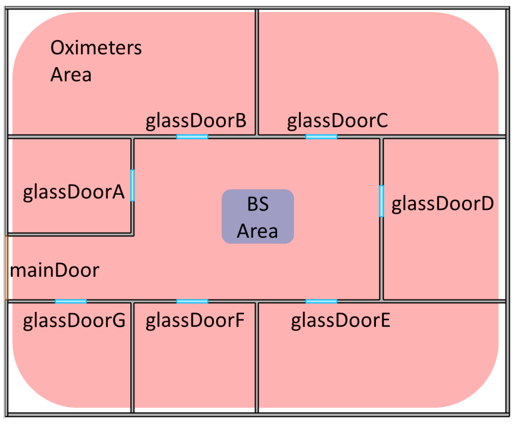

Fig. 11 Physical environment of the hospital scenario WSN

It is important to note that the above mentioned solution is one of the possible ones for deployment configurations. Another solution could also be the creation of a single area for each oximiter. Each oximeter could be placed in the centre of the area. The aforementioned solutions share the same network topology.

4.4 Mapping Modelling Language (MAPML)

MAPML is our language for assigning software components to the corresponding hardware node configuration they will be executed on. Fundamentally, a MAPML model semantically represents the classical notion of deployment of software components onto hardware resources [11]. The presence of an intermediate MAPML model between an SAML and a NODEML model helps in clearly separating the application layer from all the other lower levels of a WSN. So, architects can focus on the application from a functional point of view in SAML, while other designers can focus on low-level aspects of the WSN in NODEML. This aspect is new in the Wireless Sensor Networks research area. Figure 12 shows an overview of the metamodel underlying our MAPML language.

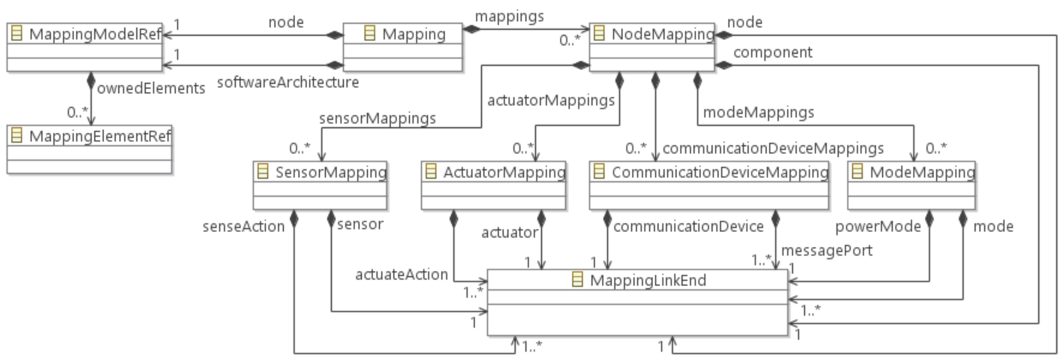

Fig. 12 MAPML Metamodel

The root of a MAPML model is the **mapping** element that references the linked SAML and NODEML models via the *softwareArchitecture* and *node* containment references, respectively. The **mapping** element is made of a set of **node mappings**, each of them linking a node definition from the NODEML model and a component from the SAML model. The semantics of a node mapping is that the linked component in the SAML model will be physically deployed on the linked node in the NODEML model. A node mapping can contain a set of secondary links, each of them can be seen as a refinement of the node mapping. Secondary links are:

- **SensorMapping** that maps either a *Sense* action or a *SensorInterrupt* event in an SAML component to a *Sensor* device in a NODEML node configuration. Fundamentally, this kind of link allows designers to specify to which physical sensor device does either a sense action or a sensor interrupt event refer to.
- **ActuatorMapping** that maps either an *Actuate* action or an *ActuatorInterrupt* event in an SAML component to an *Actuator* device in a NODEML node configuration. It is similar to the *SensorMapping* concept, but it refers to actuators, rather than sensors.

- **CommunicationDeviceMapping** that maps an SAML message port of the component linked by the parent *NodeMapping* to a NODEML radio transceiver in the node configuration linked by the parent *NodeMapping*. It allows designers to physically map a software port to its corresponding physical radio transceiver.
- **ModeMapping** that maps an energy mode defined in an SAML component to its corresponding power mode in the linked NODEML node configuration. It allows designers to decouple the two concepts of mode we have in SAML and NODEML, and thus it opens for a more flexible definition of modes in the pure "software world", independently from the power modes that the WSN node has in the "physical world".

The MAPML metamodel has some auxiliary metaclasses like **MappingModelRef**, **MappingElementRef** and **MappingLinkEnd** which are used for technical reasons . The interested reader can refer to [38] for their detailed discussion. Both the editors we developed for the MAPML language are composed of three panels that are left, centre and right. The left and right panels show the woven SAML and NODEML models, respectively, while the central panel represents various mappings of the MAPML model as a hierarchical tree. This solution allows us to provide a very clear and concise graphical editor for the MAPML model, which in some cases may have a very large number of interrelated mappings. Furthermore, we are aware that manually creating this large number of mappings can be a tedious and error-prone task for engineers; in this context, we are implementing a set of model-to-model transformations which are able to take as input an SAML model and a NODEML model, and then they are able to semi-automatically generate an initial MAPML model linking them; this operation is guided by matching strategies (e.g., name similarity via edit distance, structural similarity, etc.). In the MDE research field this practice is called *model matching* [16].

For what concerns our hospital scenario, the MAPML model linking the SAML model showed in Figure 6 and the NODEML model showed in Figure 9 has the following form:

- *NodeMapping_oximeter* links the *Oximeter* component to the *OximeterNode* node;
  - *ModeMapping_active* links both the *normal* and *alarm* modes of the *Oximeter* component to the *active* power mode of the *OximeterNode* node. It is important to note that operating modes defined in SAML are pure logical modes, whereas power modes defined in NODEML actually depend on the hardware configuration of the node itself.
  - *SensorMapping_irProbe* links both the *SenseH(h)* and *SenseH2(h)* SAML sense actions to the hardware *IRProbe* sensor in the NODEML model.
  - *ActuatorMapping_led*, which is similar to *SensorMapping_irProbe*, links both the *turn(led.GREEN)* and *turn(led.RED)* SAML actions to the hardware *led* actuator in the NODEML model.
  - *CommunicationDeviceMapping_2420* links the *update* and *alarm* SAML message ports to the *ChipCon2420* RF transceiver defined in the NODEML model.
- *NodeMapping_monitor* links the *Monitor* component to the *MonitorStation* node;
  - *ModeMapping_active* links the *normal* mode of the *Monitor* component to the *active* power mode of the *MonitorStation* node.

– *ActuatorMapping_display* links the *updateDisplay(data)* actuate SAML action to the hardware *display* actuator in the NODEML model.
– *CommunicationDeviceMapping_2420* links the *update* and *alarm* SAML message ports of the *Monitor* component to the *ChipCon2420* RF transceiver defined in the NODEML model.

With such a configuration, we now have a clear view of how various elements defined at the software architecture level interact with the hardware. For example, all the communication between the *Oximeter* and *Monitor* SAML components happen between different WSN nodes, whereas all the other actions defined in the control flow are executed locally to the component containing them. Also, the MAPML model establishes which hardware sensor and actuator equipments are actually used for performing the abstract sense and actuate actions defined in the SAML model. This level of flexibility is exactly the main goal of the A4WSN modelling approach.

4.5 Deployment Modelling Language (DEPML)

DEPML is our language for virtually deploying WSN nodes into the physical environment. Figure 13 shows an overview of the metamodel underlying the DEPML language.

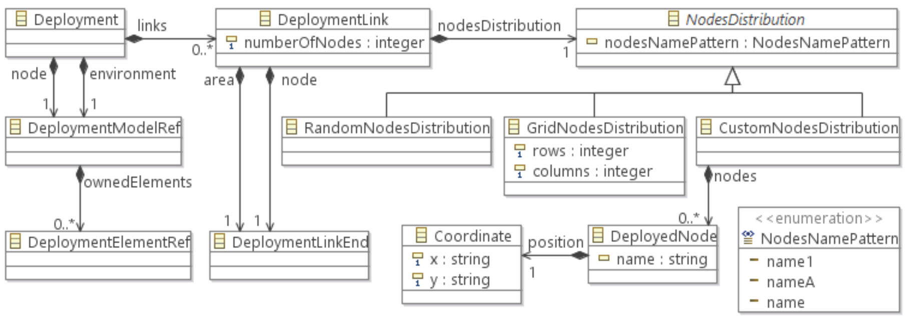

Fig. 13 DEPML Metamodel

DEPML allows designers to consider each node configuration defined in a *NODEML* model and to *instantiate* it in a specific area within the physical environment defined in a *ENVML* model. A DEPML model contains a single type of link, called **DeploymentLink** linking together a node configuration in NODEML and an area in ENVML. The semantics of the deployment link is that the linked node configuration is instantiated and virtually deployed in the linked area multiple times. This allow designers to focus on generic components and node types in SAML and NODEML, while in DEPML they can reason on the final deployment of the WSN. The number of nodes that are instantiated in the area is defined in the *numberOfNodes* attribute. Within a certain area each node configuration can be **distributed** in three different ways:

- *random*, each node is placed randomly within the area;
- *grid*, nodes are placed on a grid with a certain number of *rows* and *columns*;
- *custom*, each node is manually placed within the area. In this case, each **deployed node** is represented by its *name* (which must be unique within the area) and the coordinates of its *position*.

Also, **nodes name patterns** can be used by designers for declaring the textual pattern of the names of the nodes distributed within the area. They are used as a way to refer to the names used as targets of *Send Message* actions in SAML models. Similar to MAPML, also the DEPML metamodel has some auxiliary metaclasses like **DeploymentModelRef**, **DeploymentElementRef** and **DeploymentLinkEnd**. They are described and discussed in [38]. The DEPML modelling editor is analogous to the MAPML one. This is composed of three panels providing a tree-based representation of the NODEML, SAML and deployment links of DEPML, respectively.

For what concerns our hospital scenario, the DEPML model is very straightforward. It contains a deployment link between each node defined in the NODEML (see Figure 9) and its corresponding area in the ENVML model (see Figure 11). More specifically, the DEPML model has the following elements:

- *DeploymentLink_oximeter* links the *OximeterNode* NODEML node to the *OximetersArea* ENVML area. Since we want to specify that exactly one oximeter node must be deployed in each patient's room, we define a custom nodes ditribution. Thus, we manually define the exact position of the deployed node by means of ten *DeployedNode* elements, each of them containing the coordinates of its position in the environment.
- *DeploymentLink_monitorStation* links the *MonitorStation* NODEML node to the *BSArea* ENVML area. In this case we specify that the number of deployed node is only one, with a random distribution within the area (we can do this because the area is a square with a side of 0.5 meters, which is exactly the size of the monitoring station node).

The presented DEPML models unveil the flexibility we achieved with the A4WSN approach. Indeed, if the hospital WSN application can be reused in a different hospital, the SAML, NODEML, and MAPML models can be reused as they are. The only models that must be adapted are the ENVML model for representing the new physical environment with its obstacles and the DEPML model for linking the original NODEML nodes to the new areas, possibly with different values for specifying the number of deployed nodes (e.g., twenty oximeter nodes instead of ten).

In conclusion, all the proposed languages have been designed to provide a good trade-off between genericity, expressivity and accuracy in capturing the various facets of the WSN domain. To this respect, it is fundamental to allow designers to check whether their models are correct with respect to the semantics of the proposed languages. More specifically, A4WSN allow designers to check whether a model adheres to the structural semantics of its corresponding language (e.g., SAML). A4WSN supports this feature by leveraging the well-known notion of **conformance** in Model-Driven Engineering; in other words, in A4WSN a model m adheres to the structural semantics of its corresponding language (e.g., SAML) if and only if m actually conforms to its metamodel (e.g., the SAML metamodel introduced in Section 4.1). Furthermore, in order to ensure a more precise semantics of the languages described in Section 4, we complemented them with fourteen OCL[8] constraints. For example, in NODEML a constraint ensures that each instance of *Node* must contain at least one program memory, another constraint in DEPML ensures that the coordinates of each manually positioned node must be within the boundaries of the area it is deployed in, and so on. For the sake of readability the description of such constraints are not discussed extensively. If the need for more strict semantics of the proposed languages arises (for instance in order to define WSN applications with specific styles or special configurations), additional OCL constraints can be added to every element of the languages by extending the A4WSN platform with a suitable plugin. Please, refer to Section 5 for more details on this feature of the A4WSN platform.

5 The Programming Framework

As introduced in the beginning of Section 3, the A4WSN platform is composed of two main parts that are a modelling environment to allow architects to model WSN applications and a programming framework devoted to code generation and analysis of WSN application models. The motivation for performing code generation and analysis of WSN application models are well understood both in academia and in practice [51,36]. Basically, code generation helps in reducing the cost of developing a WSN application since the developers can automatically obtain an executable application from the model by applying some specific transformations. Also, performing analysis is fundamental while developing a WSN application due to the intrinsic complexity of the WSN domain. For example, if we consider typical aspects in WSN development such as nodes connectivity, real-time communication, energy consumption, performance, security, etc., it is extremely difficult and demands for a lot of effort to ensure that a developed WSN is correct with respect to those aspects. Moreover, analysis engines can also be used to reason on the WSN configuration in order to find reasonable trade-offs in terms of network topology, employed protocols, etc. for a specific task.

In this section we present the *generic and extensible* programming framework of A4WSN. It is tailored to support the development of code generation and analysis engines against WSN application models conforming to the modelling languages described in Section 4.

Our programming framework provides a generic workbench and a set of extension points for supporting the development and integration of third-party code generation and analysis engines. More specifically, through its components, it enables the storage of WSN models, supports the merging of linked models, validates A4WSN models, provides error/warning/information messages to the user, defines a UI manager to make plugins interacting, provides facilities for managing code generation and analysis engines.

[8] Object Constraint Language (OCL) specification: http://www.omg.org/spec/OCL/2.3.1

Third-party engines are realised as plugins extending the A4WSN generic workbench. It knows at run-time which plugins are available and automatically provides to the user the available target implementation languages and analysis techniques.

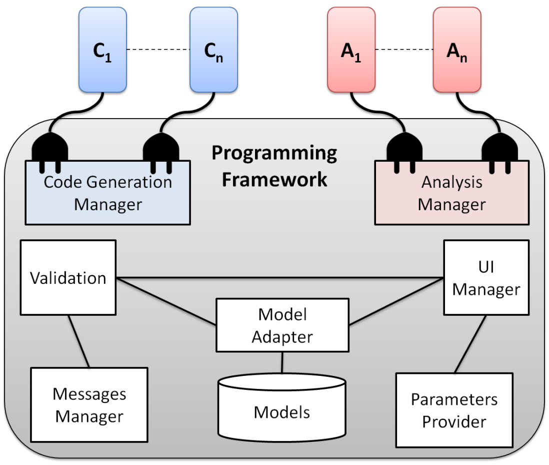

Fig. 14 The A4WSN programming framework

Figure 14 shows an overview of the A4WSN programming framework. All the boxes within the programming framework represent the various components of the generic programming workbench, whereas the $C_1..C_n$ and $A_1..A_n$ boxes represent third-party code generation and analysis plugins, respectively. Third-party plugins extend the *Code Generation Manager* and *Analysis Manager* components which provide the needed extension points and they communicate with all the other components of the programming framework (for the sake of clarity we do not show those connectors in the figure). In the following we will discuss the facilities and duties of the various components of the generic A4WSN programming framework, an overview of their implementation details is provided in Section 6.

5.1 Models

The central element of the programming framework is the Models repository that stores all the WSN models developed by architects and designers. Indeed, stored models can conform to any modelling language described in Section 4 which are SAML, NODEML, ENVML, MAPML, and DEPML. The models repository can be realised in different ways. For instance it may directly rely on the file system of the machine running the A4WSN platform (this is the solution implemented in the current version of the A4WSN tool), it may point to resources stored in the cloud or it may refer to some in-memory models representation. If on one side this feature of the models repository is very flexible in terms of resources consumption and localisation, on the other side it opens for possible problems of interoperability between all the other components of the A4WSN programming framework. This is exactly why the Model Adapter component exists.

5.2 Model Adapter

The model adapter is a component which abstracts the nature of the models repository to the other components of the A4WSN programming framework. The model adapter is composed of a set of connectors (each of them tailored to a specific models storage type) that expose a common interface to all the other components to access various elements of the models in a homogeneous way. Also, the Model Adapter component has a built-in model transformation, called *Merger*, that can merge linked models defined in the A4WSN modelling environment. If we consider *Merger* as a function, it can be defined as follows:

$$Merger\text{: } MM_{SAML}\ x\ MM_{NODEML}\ x\ MM_{ENVML}\ x\ MM_{MAPML}\ x\ MM_{DEPML} \rightarrow MM_{merge}$$

where each MM_x is the metamodel of the x modelling language, where x can vary between $SAML$, $NODEML$, $ENVML$, $MAPML$, $DEPML$, and MM_{merge} is the union of all the MM_x metamodels. In other words, *Merger* takes as an input an instance of each modelling language defined in the A4WSN modelling environment and provides a single model conforming to a unique metamodel as an output. The reason behind the existence of the Merger transformation is that currently many approaches and tools for code generation and analysis assume to have a single model as an input, rather than a set of models conforming to different languages. In order to alleviate this issue with current approaches and tools (which could have hampered the usefulness of the whole A4WSN platform), we decided to implement the Merger as an internal transformation to merge separate models into a single one. Merger can be executed at any time by plugin developers by calling a dedicated Java method.

5.3 Validation

The Validation component executes all the operations to validate A4WSN models:

- it checks whether one of the A4WSN models conforms to its corresponding metamodel (metamodels are described in Section 4);
- it executes all the OCL constraints defined in each metamodel within the A4WSN platform and checks whether they are satisfied or not;
- if defined, it executes the additional OCL constraints that are defined in some code generation or analysis plugin and checks whether they are satisfied or not.

The result of a validation operation is composed of four main elements: (i) a boolean value representing whether the involved model passes all the checks listed above, (ii) a set of informative messages that describe the result of the validation in a human-readable way, (iii) a set of in-memory representations of all the elements in the models which do not satisfy some of the checks listed above, and (iv) a set of actions that can be executed by the A4WSN platform as a quick fix of the identified violations (quick fix operations can be defined in the plugins extending the A4WSN platform).

The Validation component communicates with Model Adapter in order to access various elements of the models to be validated. Also, it communicates with the Messages Manager and the UI Manager components to show the informative messages belonging to M to the user and to highlight the elements in their graphical editor violating the constraints, respectively.

5.4 Messages Manager

The Messages Manager component serves to graphically show informative messages to the user. A4WSN supports three kind of informative messages which are *error, warning and information*. Plugin developers can decide the type of each message to be shown, depending on its severity. Each message is defined as a couple $< K, T >$, where K represents the type of message (i.e., error, warning, or information) and T represents the textual content of the message in a human readable way.

5.5 UI Manager

The UI Manager component is responsible for the main facilities interacting with the user interface of the A4WSN platform[9]. The UI Manager component provides all the graphical facilities to interact with the plugins and elements of the A4WSN platform, which are:

- *Code Generation Engines View*: a dedicated view showing a list of all the available code generation engines (with their description, icon, name, etc.), together with their management facilities, such as code generation activation, code generation results viewer, etc.;
- *Analysis Engines View*: a dedicated view showing a list of all the available analysis engines (with their description, icon, name, etc.), together with their management facilities, such as analysis activation, analysis results viewer (significantly different from the code generation results viewer), etc.;
- *Code Generation Contextual Menu*: a contextual menu that triggers the execution of a code generation engine. A contextual menu is associated to each model of the A4WSN modelling environment;
- *Analysis Contextual Menu*: a contextual menu that triggers the execution of an analysis engine. A contextual menu is associated to each model of the A4WSN modelling environment;
- *Validation Trigger*: a contextual menu and a dedicated button in the graphical editor of each model of the A4WSN modelling environment that triggers the validation of the current model. Optionally, the user can identify which plugin contains additional constraints to be checked. The results of the triggered validation are managed by the Messages Manager component;

[9] Also the Messages Manager interacts with the UI of the A4WSN platform, however its impact to the UI is much more limited than that of UI Manager.

– *Code Generation and Analysis Progress Feedback*: provides an element in the UI that graphically shows the progress of the triggered code generation or analysis. A4WSN provides two types of progress feedback, a progress bar for activities in which all the steps are known a priori and a round indicator for activities with an unknown length.
– *Plugin Additional Parameters View*: provides a dedicated view in which users can provide additional parameter to be passed to the code generation or analysis engine being triggered. Plugin developers can specify the number, name, and type of those parameters by using a specific extension point.

5.6 Parameter Provider

Parameter Provider component manages the additional parameters that a code generation or analysis plugin may require for carrying on its activities. As previously mentioned, additional parameters are defined by using a specific extension point of the A4WSN programming framework; each parameter is defined as a triplet $< name, T, default >$, where $name$ is the unique name of the parameter, T is the type of the parameter, and $default$ is the optional default value of the parameter. Available parameter types are listed below.

– *String*: a textual value;
– *Integer*: an integer numerical value;
– *Float*: a decimal numerical value;
– *Boolean*: a boolean value;
– *Local Resource*: a file in the local file system of the user, it is referenced by its path in the file system;
– *Remote Resource*: a resource in the cloud that can be accessed by a standard HTTP GET request and is referenced by its URL.

Once the user has provided the values of the additional parameter of a code generation or analysis engine, the Parameter Provider component makes them available to the plugin realizing the engine so that it can access them before actually executing the activity which is being triggered by the user.

5.7 Code Generation Manager

The Code Generation Manager provides a set of facilities for managing code generation engines and the extension point that is used by code generation plugin developers (see Section 5.9 for more details). For instance it checks which plugins are currently extending its extension point and makes their facilities available to the end user. It includes all the registered code generation plugins into the *Code Generation Engines View* of the UI Manager. It loads plugins into the contextual menus of the A4WSN modelling environment. It automatically triggers the validation operations defined by the plugins before executing the actual code generation operation. Also, the Code Generation Manager component exposes a common *Java API* to plugin developers, so that they can easily interact with all the other components of the A4WSN programming framework. For example, it allows developers to access elements of the models in the Models Repository to push messages to the end user via the Messages Manager and it makes the additional parameters provided by the end users accessible directly as Java objects.

5.8 Analysis Manager

The internal logic of the Analysis Manager component is analogous to that of Code Generation Manager. The only difference is that it is designed for analysis plugins, rather than for code generation plugins. Due to its similarity to Code Generation Manager, the reader can easily grasp its functioning from the description of the latter, so we will not describe the Analysis Manager component in this paper. In Section 5.9 we discuss the extension points that are available to code generation and analysis plugin developers.

5.9 Extension Points

The concept of extension point is nicely described in the Eclipse Wiki[10], it says that *the extension point declares a contract, typically a combination of XML markup and Java interfaces, that extensions must conform to. Plug-ins that want to connect to that extension point must implement that contract in their extension. The key attribute is that the plug-in being extended knows nothing about the plug-in that is connecting to it beyond the scope of that extension point contract. This allows plug-ins built by different individuals or companies to interact seamlessly, even without their knowing much about one another.* The last part of the Eclipse definition of extension point says exactly what we are demanding to the WSN research community, i.e., not to rebuild the wheel by focussing on modelling languages, graphical editors, etc., but rather to focus on code generation and analysis of WSN applications by developing A4WSN plug-ins.

Table 1 shows various extension elements that can be set by third-party developers with their plugins. For each element we specify its name, whether it belongs to the code generation (column titled *CG*) or analysis extension point (column titled *A*), and a description about how it will be used by the generic A4WSN programming framework.

The extension points defined in the A4WSN programming framework are used to group code generation and analysis engines into two different groups, so that the end user knows where those engines can be found. Also, they are used to provide a common, standard behaviour to various engines that may be defined upon the A4WSN modelling environment. Both the Code Generation Manager and Analysis Manager provide a standard management of the workflow that must be followed when executing those engines. For example, they automatically call the pre-actions defined by using the *Pre Action* element of the previously defined extension point (the same

[10] http://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points\%3F

Element	**CG**	**A**	**Description**
Name	✓	✓	The name of the engine being provided which will be shown in the engines view and contextual menus.
Icon	✓	✓	An image icon of the engine being provided which will be shown in the engines view and contextual menus.
Description	✓	✓	A textual description engine being provided which will be shown in the engines view.
Network Access	✓	✓	A boolean for declaring whether the engine uses the network for its operations which will be shown in the engines view.
Operation Time	✓	✓	An estimation of the time needed to complete the operation being defined which will be shown in the engines view.
Target Languages	✓	-	A list of the target implementation languages which will be shown in the engines view and contextual menus.
Target Path	✓	-	The path in the file system (local to the location of the plugin) to which the generated code will be saved.
Analysis Type	-	✓	A list of the properties that will be checked during the analysis operation (e.g., performance, security, etc.); it will be shown in the engines view.
Keep Intermediate	-	✓	A boolean value (optionally a path in the file system) to specify whether (and where) the analysis engine keeps possible intermediate resources.
Additional Parameters	✓	✓	A list of parameter types definition that will be used by the Parameter Provider component of A4WSN.
Validation Constraints	✓	✓	A list of OCL constraints, together with their informative messages and quick fix operations that must be used by the Validation component of A4WSN.
Pre Action	✓	✓	A reference to a Java class defining the method that will always be called before executing the engine being provided.
Post Action	✓	✓	A reference to a Java class defining the method that will always be called after the engine being provided is executed.

Table 1 Elements of the extension points for code generation or analysis plugins

holds for the post-action). Automatically manage the success and error messages to be shown after the execution of either a code generation or analysis operation, automatically update the UI of the modelling framework depending on the available plugins extending A4WSN, etc. Moreover, since plugin developers must comply with to the extension points defined in the A4WSN programming framework, they will be more keen to provide engines that are straightforward to integrate and with common basic functionalities, thus easier to use by end users. Section 7 describes an example of plugin for estimating the energy consumption of a WSN, it will be applied to our case study in the health care domain.

6 Implementation

We make the current prototype of the proposed approach available to the community as an open-source product with MIT license in order to allow other researchers to use the modelling languages introduced in Section 4 as well as the programming framework described in Section 5. The current prototype of A4WSN can be downloaded from the A4WSN website (http://a4wsn.di.univaq.it).

We implemented the proposed approach by extending the **Eclipse** platform[11]. Eclipse is an open-source development platform comprised of extensible frameworks and tools for building, deploying and managing software across the life cycle. We decided to use Eclipse as starting point for our modelling environment for three main reasons. First, many extensions already exist covering some aspects of our approach (e.g., graphical syntax definition for newly created languages, models persistence support, etc.). Second, its plugin architecture allows us to provide a set of extension points that other developers can use to extend our modelling framework,. Third, the Eclipse community is widely spread throughout the world, raising the possibility of adoption of our modelling environment.

For what concerns the modelling languages, model-driven engineering techniques are used to define their concepts, and their modelling environment. More specifically, we specified the static semantics of the languages by means of their underlying metamodels. Those metamodels are defined by using the **Eclipse modelling Framework** (EMF)[12], that is a Java framework and code generation facility for building tools and other applications based on a metamodel. The concrete syntax of the modelling languages has been defined by using the **Graphical modelling Framework** (GMF)[13], a model-driven approach to generate graphical editors in Eclipse.

The intermediate modelling languages (i.e., MAPML and DEPML) are technically called weaving models. Weaving models are special kinds of models for defining relations among other models and to establish semantic links among model elements. Weaving models have been successfully used in many fields, such as software architecture [41] and software product lines [13]. We use the **Atlas Model Weaver (AMW)** [18] for managing those weaving models.

For what concerns the programming framework, we implemented it as a set of **Eclipse plugins**, each one implementing a single component of the programming framework, as it is depicted in Figure 14. Those plugins are implemented in Java and their dependencies are realized by means of the plugins management system provided by Eclipse. Each plugin declares the others it depends on and configuration parameters via a specific XML configuration file. The communication among plugins is handled by standard Java calls. Also, the code generation framework and the analysis framework provide two extension points dedicated to code generation and analysis plugins, respectively. The signatures of those extension points are defined in the same XML configuration files used for defining the dependencies between plugins, whereas their implementation is defined as Java classes referenced by the XML configuration files. For the sake of brevity, we do not provide the details on how the programming framework works and on how its plugins interact. A detailed description of those aspects can be found in the Eclipse plugin developer guide[14].

[11] Eclipse project Web site: www.eclipse.org.
[12] EMF project Web site: http://www.eclipse.org/modeling/emf/.
[13] GMF project Web site: http://www.eclipse.org/modeling/gmf/.
[14] Eclipse Platform Plug-in Developer Guide: http://help.eclipse.org/helios/index.jsp

7 PlaceLife: an A4WSN plug-in

In order to validate the expressivity of the A4WSN modelling languages and to exercise the provided extension points, we developed an analysis plug-in called PlaceLife. PlaceLife takes advantage of the three modelling views (namely, SAML, NODEML and ENVML) in order to provide an estimate of the WSN lifetime. All modelling views are analysed, combined and translated into low level simulation scripts that can be executed to estimate the WSN lifetime. This translation has been useful to verify that our models have an appropriate level of details for simulation purposes. In order to produce a realistic simulation the following is desirable:

- **abstraction:** the models abstract all the details needed to generate scripts that can run in various well-accepted simulators such as Opnet and OMNET++;
- **fine-grain simulation:** the details should allow fine-grain simulations that combine different information such as physical environment, hardware and various layers of OSI.

We verify abstraction by considering all OSI layers and for each layer the information required by well-established simulation tools. We show that in most of the cases our models abstract the information but missing ones can be easily added. This is done from Section 7.1 to Section 7.3.

Fine-grain simulations are easily obtained thanks to the reuse and the weaving of multiple models into a single one. Models such as NODEML or ENVML that contain low level information (e.g., hardware and path loss) can be created once and reused multiple times with different application models. While expert users can write these complex low level models, average users can reuse them as many times as they like and concentrate on the application logic. This has been validated in Section 7.6 by means of a home automation application. Technical details such as the effects of the path loss and the hardware which require theories of telecommunication are specified in pre built PlaceLife models. Technical details are complemented with the application model (i.e, SAML) and the physical environment (i.e., ENVML). These models are transformed into various complex simulation scripts. In Section 7.5 we describe the PlaceLife implementation and the simulation tool used as a target language for simulation script generation (that is, Castalia). In section 7.6 we compare the simulation results obtained with a basic Castalia simulation written by an average user with a PlaceLife simulation based on pre-built hardware and path loss models, applied to the fire alarm and automatic heating application. Numerical results are also presented to show the effects of realistic simulation scenario, where environmental factors are taken into account. The former has the default ideal free space model for the path loss while the latter considers pre-built PlaceLife models that consider the real environment that is made of physical objects. We see that not considering the real environment may cause overestimation of the lifetime which is particularly undesirable.

7.1 Application layer

Information at application layer should include the structure and behaviour of the WSN. For instance this includes type of components, number of instances and their interaction. This information is useful to derive relevant data such as the sensing rate, messages sent over the network and the type of communication (broadcast, multicast and point-to point). This data clearly affects the energy consumption. Beside the structure and the behaviour of the WSN, useful application layer information can be the used aggregation protocol[15] [31,52] (if any), the type of operating system, the type of middleware (if any) and so on.

The modelling languages of A4WSN provide ways to define all the aforementioned application layer information. The SAML view contains structure and behaviour of the WSN application. For instance this includes the type of components, their interaction, the sensing and transmission activities of a node and the type of transmission. This information is complemented by the NODEML view that specifies, among the other information, the type of operating system and the middleware used. In PlaceLife we use the ENVML and DEPML models in order to have data about the number of nodes within the WSN and their deployment position in the environment. Application layer information is translated into low level scripts. More precisely, structure and behaviour are translated into simulation scripts. These scripts are combined with components from the simulation library (such as sensing components and middleware) in order to obtain the entire application layer configuration of the simulation. While PlaceLife provides the implementation of some libraries, unimplemented ones such as unknown sensors or unsupported middleware need to be specified by the user.

7.2 Networking and data link layers

Information at networking layer should specify the routing protocol. While routing can be performed by using a multi-hop solution, clustering approaches are very effective in order to improve the energy efficiency of the WSNs. This is why the NODEML can specify either multi-hop routing protocols (e.g., AODV) or some clustering approach (e.g., LEACH). A static routing can also be defined by explicitly specifying the connection among nodes. Routing that are not supported by A4WSN must be implemented by using some simulation script language.

Information at data link layer should include the medium access method (MAC) that is used. Access methods can be summarised into two main categories: contention based method (e.g., CSMA/CA) and channel partitioning (e.g., TDMA). The NODEML includes a wide range of possibilities for the MAC protocol selection. This includes CSMA, T-MAC and S-MAC [70,14].

[15] Aggregation and fusion aims at removing redundant data and transmitting concise information.

7.3 Physical layer and hardware

Physical layer information should support the definition of an energy consumption model for realistic estimate of the WSN lifetime. An advanced energy consumption model should consider the path loss, the modulation scheme, the hardware that is used, the coding scheme and so on. While the modulation scheme, the hardware and the coding scheme are specified in the NODEML model, the path loss, as we show in the next section, has been defined according to the environment and its obstacles. NODEML, ENVML and path loss definitions are used to generate low level settings and scripts that can provide a fine estimate of the energy required to transmit a bit over the physical channel.

7.3.1 The path loss

In sensor networks, path loss can play a crucial role since neglecting the path loss may cause overestimation of WSN lifetime. The optimistic evaluation of resources is particularly dangerous for WSNs since the resources come with significant restrictions especially in terms of energy. The path loss is reduction in transmitted signal strength as a function of distance, which determines how far apart two sensor devices can be and have reliable communication between the devices [49]. The core of signal coverage calculations for any environment is a path loss model, which relates the loss of signal strength to the distance between two terminals and the operating frequency.

Indoor radio propagation is dominated by the same mechanisms as outdoor propagation: reflection, scattering, diffraction, refraction, absorption and depolarization. However, conditions are much more variable. The indoor environment differs widely due to the increased number of obstacles, layout of rooms, presence of multiple walls and floors, windows and open spaces. Altogether these factors have a significant impact on path loss in an indoor environment. Due to the irregularity in the position of obstacles and layout of the rooms, the channel varies significantly with the environment making the indoor propagation modelling relatively inconsistent and challenging especially for modelling. The propagation and path loss models are usually based on empirical studies on the system considered.

Accurate modeling of the actual environment is very complex as the communication systems operate in complex propagation environments. In practice, most of the simulation studies make use of the empirical models that have been developed based on empirical measurements over a given distance in a given operational frequency range and a particular environment [53]. Some of the most common empirical models include Okumura Model, Hata Model, COST 231-Walfish-Ikegami Model, Erceg Model, ITU Indoor Path Loss Model, Log-Distance Path Loss Model etc [53,60]. When considering an indoor propagation environment for path loss, the material used for walls and floors, the layout of rooms, windows and open areas, location and materials obstructing etc. should be taken into account, as all of these factors have a substantial impact on the path loss in an indoor environment. The complexity of signal propagation in the indoor environment makes it difficult to obtain a single model that illustrates path loss across a wide range of environments. The following is a commonly used simplified model for path loss as a function of distance [24].

Table 2 Partition dependent losses for 2.4 Ghz

Attenuating Material	Signal attenuation in dB
Wood	2
Metal frame, glass wall into building	6
Office wall	6
Metal door in office wall	6
Cinder wall	4
Metal door in brick wall	12.4
Brick wall next to metal door	3

$$P_r\ (dBm) = P_t\ (dBm) + K(dB) - 10\gamma\ log_{10}\left(\frac{d}{d_0}\right) \tag{1}$$

where, K is the path loss factor, which depends on antenna characteristics and the average channel attenuation. γ is the path loss exponent, d_0 is reference distance for the antenna far field, and is typically assumed to be 1-10 m for indoor scenarios and 10-100 m for outdoor scenarios. P_r is the received signal strength and P_t is the transmitted signal strength. K can be calculated as:

$$K\ (dB)\ =\ 20\,log_{10}\frac{\lambda}{4\pi d_0} \tag{2}$$

The value of λ depends on the propagation environment. This path loss model, together with the ENVML physical environmental model, is used to define the path loss between any two nodes. Please note that existing simulation packages, and modelling architectures do not consider the effects of path loss to best of our knowledge. We fix the value of γ at 2 for free space and introduce the losses for each partition (obstacle) that is encountered by a straight line connecting the receiver and the transmitter. Please refer to Table 2 for the decibel loss values measured for different type of partitions, at 2.4 GHz [49]. In order to add the effects of obstacles between the transmitter and the receiver, we add the fixed path losses per existing obstacles to the free space path loss.

There is no doubt that the physical layer has a fundamental role when energy consumption is considered. Researchers are currently investigating various modulations and coding techniques. Choosing a model for energy consumption can also be complicated. Different studies and simulation tools [27] [9] [66] consider different models for energy consumption.

7.4 Analytical model

In this section the analytical framework for calculation of the lifetime of nodes is presented. The analytical model presented is in turn used to verify the simulation results. The analytical framework consider the characteristics of the transmission process which affects the lifetime of a node. Without the loss of generality the lifetime of a node can be expressed as:

$$LifeTime_{node} = \frac{E_{tot}}{E_{pr} \times Rate_r + E_{pt} \times Rate_t} \tag{3}$$

where $LifeTime_{node}$ is the lifetime of the nodes, E_{tot} is the total energy in joules available for the node considered (e.g. initial energy for two AA battery is 18720 joules), E_{pr} and E_{pt} are the energy spent to receive and transmit a single packet, and $Rate_r$ and $Rate_t$ are the average number of packets received and sent per second respectively.

For the analytical framework introduced in this paper the energy consumption model described in[9] is combined with the effective number of transmissions including the retransmissions caused by obstacles. This model is selected since it includes various factors such as distance, attenuation due to obstacles, modulation, and hardware. The energy consumption of a node n is affected by the following three terms:

– E_{ct}: the transmitter circuit energy;
– E_{cr}: the receiver circuit energy;
– E_t: the transmission energy.

The calculation of the transmission energy E_t is based on the following expression:

$$E_t = \frac{LMN_fN_0}{\gamma G} \times f_{\tau,\Im}(B) \tag{4}$$

where L is the path-loss (see [35] for details), M is the link safety margin, N_f is the receiver noise figure, N_0 is the ambient noise power spectral density, γ is the power amplifier efficiency, G is the combined gain of the transmit and receive antennas, $f_{\tau,\Im}(B)$ is the required signal-to-noise ratio per bit corresponding to transmission technique τ, fading characteristics $\Im$ and target bit error rate B.

The energy spent for receiving a packet can be calculated as follows:

$$E_{pr} = (E_{cr}) \times packetsize \tag{5}$$

Table 3 Selected values

link safety margin	M=10
receiver noise figure	N_f=5
ambient noise power spectral density	$N_0 = -204$dBJ
power amplifier efficiency	γ=0.35
combined gain of the transmit and receive antennas	G = 1
required signal-to-noise ratio per bit τ=transmission technique $\Im$= fading characteristics and B=target bit error rate	$f_{\tau,\Im}(B)$ = 15dB
circuitry	$E_{ct} = E_{cr} = 1\mu J$

7.5 PlaceLife implementation

PlaceLife is a code generation plugin that generates Castalia and OMNET++ [47] simulation scripts. Castalia is a WSN simulator based on the OMNeT++ platform. It is mainly used for initial testing of protocols and/or algorithms with a realistic node behaviour, wireless channel and radio models. The OMNeT++ platform is an extensible, modular, component-based C++ simulation library and framework, primarily for building network simulators.

Castalia is used in order to simulate the radio channel, the MAC protocol and the network protocol. The application behaviour is needed to derive application level simulation parameters. The environment and the path loss models allow the calculation of the path loss. In fact, while Castalia assumes that the user provides path loss related parameters in a complex path loss matrix, PlaceLife presents an abstract view of the environment where the path loss is derived based on the characteristics of the environment specified in the ENVML model. OMNET++ is used for additional simulation components such as the sensing devices and the middleware library.

7.6 PlaceLife applied to the home automation system: Numerical Results and Discussions

In order to show the effectiveness of the architectural approach and the PlaceLife plugin, in this section (i) we consider a case study about a home automation system, and (ii) we present numerical results of its simulation. The numerical results also show the effects of realistic simulation scenarios, where environmental factors are taken into account.

Monitoring and automatic control of building environment is a case study considered quite often [26], [22]. Home automation can include the following functionalities: (i) heating, ventilation, and air conditioning (HVAC) systems; (ii) emergency control systems (fire alarms); (iii) centralised lighting control; and (iv) other systems, to provide comfort, energy efficiency and security. In order to validate our approach we consider the fire alarm system and the automatic heating application. A CC2420 chip, compatible with 802.15.4, is used to provide wireless communication, operating at 2.4 GHz and providing a maximum data rate of 250 kbps. The transmission output power with which the radio transmits the packets is 0dBm. It employs Direct Sequence Spread Spectrum (DSSS) modulation in combination with Offset - Quadrature Phase Shift Keying (O-QPSK) modulation. Each node is powered by two AA batteries with up to 18720 Joules and uses a Texas Instruments ChipCon 2420 RF transceiver. Low-power Atmel AVR ATmega128 equipped with an ADC is used as the micro controller. The fire alarm system is composed of temperature sensors, smoke detectors and sprinkler actuators. In our fire alarm implementation we assume that all the temperature sensors monitor the temperature at regular intervals Δt_1. When a temperature sensor reads a value that exceeds a specified threshold T and a smoke sensor detects smoke all the sprinklers are activated. The value Δt_1 and the threshold T are assumed to be 30 seconds and 50 celsius degree, respectively.

The automatic heating application is composed of different temperature sensors, a base station, and various heaters. In our automatic heating application the temperature sensors send readings at regular intervals Δt_2 to the base station directly (no routing protocol is employed as the sensors communicate directly to the base station). This is placed at the center forming a star topology. The base station averages the readings and decides whether or not the central heating system should be on. More specifically the base station works in the following way:

– if the heating is turned on and the average temperature is greater than the maximum temperature T_{max}, the central heating system turns off.
– if the average temperature is less than the minimum temperature T_{min}, the central heating system turns on.

The value Δt_2 is set to be 30 seconds while $T_{min} = T_{max} = 22$ Celsius degree. We assume the fire alarm system and the automatic heating application are deployed in a building composed of three floors. Each floor has the same floor plan that is shown in Figure 15. It is important to note that base stations interference is negligible since base stations have no energy limitations and they do not use different channels.

Figure 15 represents the floor plan of an apartment, containing the temperature and smoke detector nodes. For the sake of representation, we use numbers to represent sensor nodes monitoring temperature and smoke, and we do not present the various models representing the WSN, rather we directly describe the main results of the execution of the automatically generated simulation scripts. Nodes 1, 2 represent the temperature and smoke detector nodes respectively in the bathroom area. Nodes 3,4 respectively represent the temperature and smoke detector nodes in the pantry. Nodes 5, 6 represent the temperature and smoke detector nodes respectively in the kitchen, nodes 7,8 represent temperature and smoke detectors respectively in the storage area while nodes 9, 10 represent the nodes from the outer hallway and stairs to the lobby. All these temperature nodes in these areas sense and send to the base station located close to the pantry.

In the larger bedroom, nodes 11, 12 represent the temperature and smoke detectors nodes respectively, nodes 15, 16 represent temperature and smoke detector nodes in the living room, while 17, 18 represent temperature and smoke detector nodes in the smaller bedroom. In the closet attached to the smaller bedroom, nodes 19, 20 represent temperature and smoke detector nodes respectively. Nodes 21 and 22 respectively represent temperature and smoke detector nodes in the lobby. Nodes 13, 14 are temperature and smoke detector nodes respectively, located close to the base station in the living room. All the temperature nodes in these areas sense and send their data to the base station in the living room. The base stations are connected to each other using peer-to-peer connection. Coloured representation of the signal strength degradation due to obstacles can be seen in the Figure 15. Coloured representation is used to clearly show the effect of signal strength due to obstacles in the home environment. Yellow colour is used to represent the wooden obstacle, while glass is represented by blue colour and red is used to represent the walls in the home environment. For the areas not affected by path loss due to obstacles is shown in white coloured background.

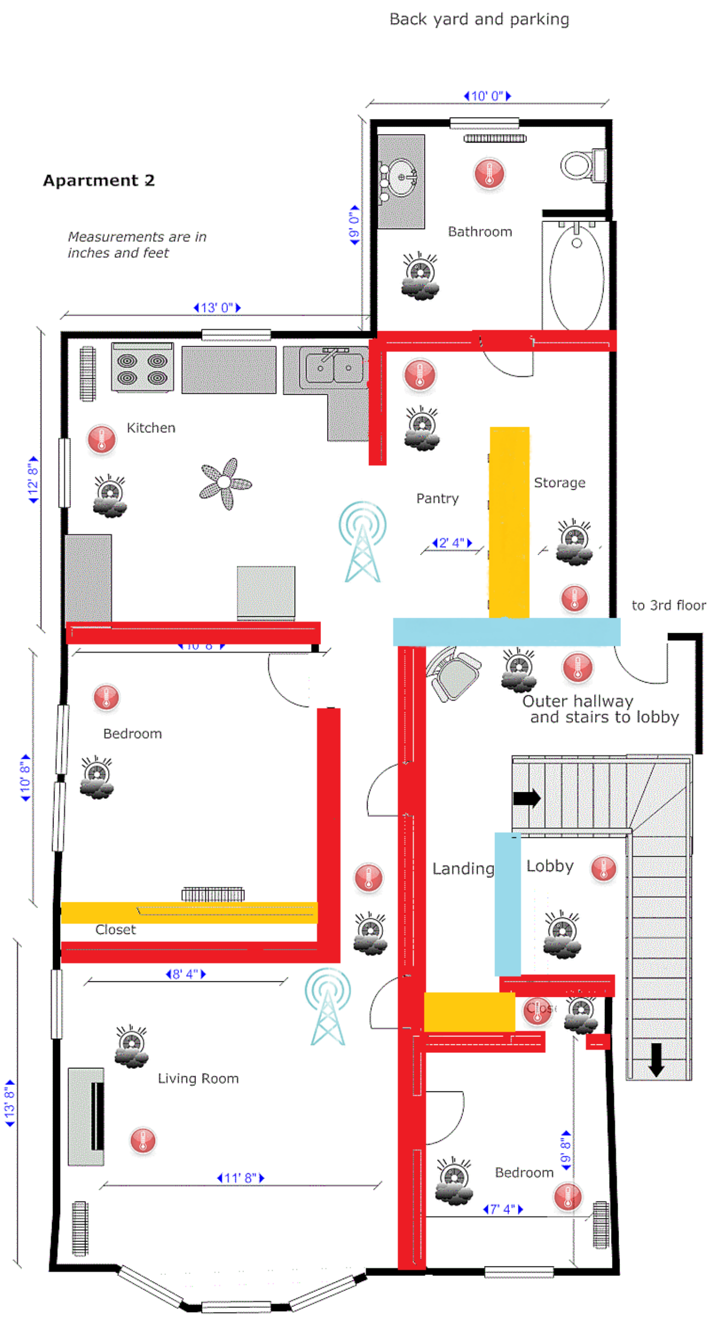

Fig. 15 Home automation - case study

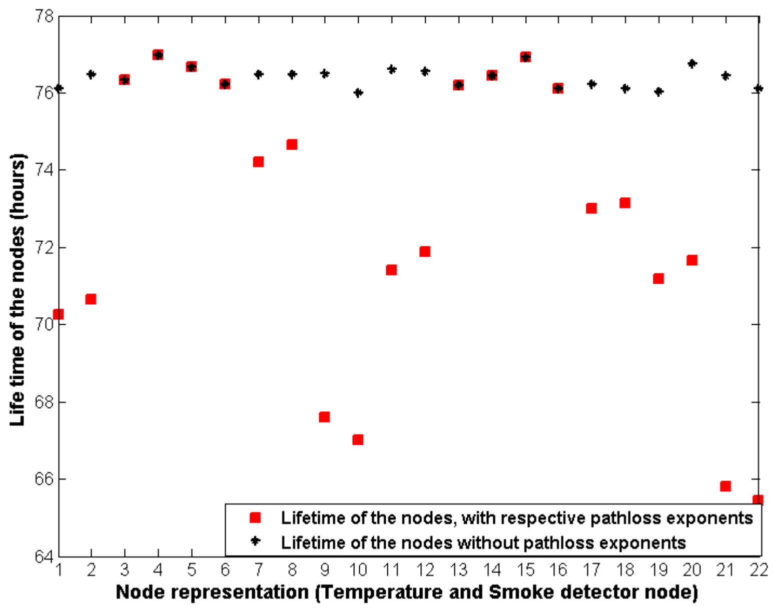

Fig. 16 Life time of the nodes

Figure 16 shows the energy consumption of each node in one of the floors of the building for free space environment and when the path loss due to obstacles (mainly due to partitions) is introduced. It is evident that ignoring the effect of path loss would be an optimistic assumption when energy consumed by each node is considered. The obstacles considered are mainly the wall partitions used for indoor segmentation. The results clearly show that avoiding path loss would cause overestimation of the WSN lifetime. More precisely, the lifetime of the nodes 1 and 2 deployed in the bathroom area is about 76 hours with no path-loss as compared to 70.5 when the exponent due to the brick wall separating the bathroom and the pantry are considered. Similarly, the lifetime of the nodes 21 and 22 is about 65.5 hours when the attenuation due to the glass partition in the lobby area and also the brick wall separating the landing area and the living room, as compared to 76 hours ignoring the effects of path loss. It can also be observed that the nodes 3, 4, 5, 6, 13, 14, 15, 16 are not affected by path-loss as they are not enclosed by walls or any obstacles. Hence, their lifetime is roughly about 76.5 hours.

It is evident that ignoring the effect of path loss would be an optimistic assumption when energy consumed by each node is considered. Results presented in Figure 16 are particularly important to show the usefulness of a detailed and a realistic modelling tool. Our PlaceLife plug-in allows engineers to consider the nature of the obstacles of the environment in details, thus providing a more realistic performance measurement. Our PlaceLife plugin allows engineers to consider the nature of the obstacles of the environment in details, thus providing a more realistic performance measurement. While various components of PlaceLife are employed, the design of the simulation does not get complicated since the architecture presented is user friendly. Please note that the multi-view architectural approach allows the user to isolate the physical environment an incorporate various factors such as path loss, shadowing etc.

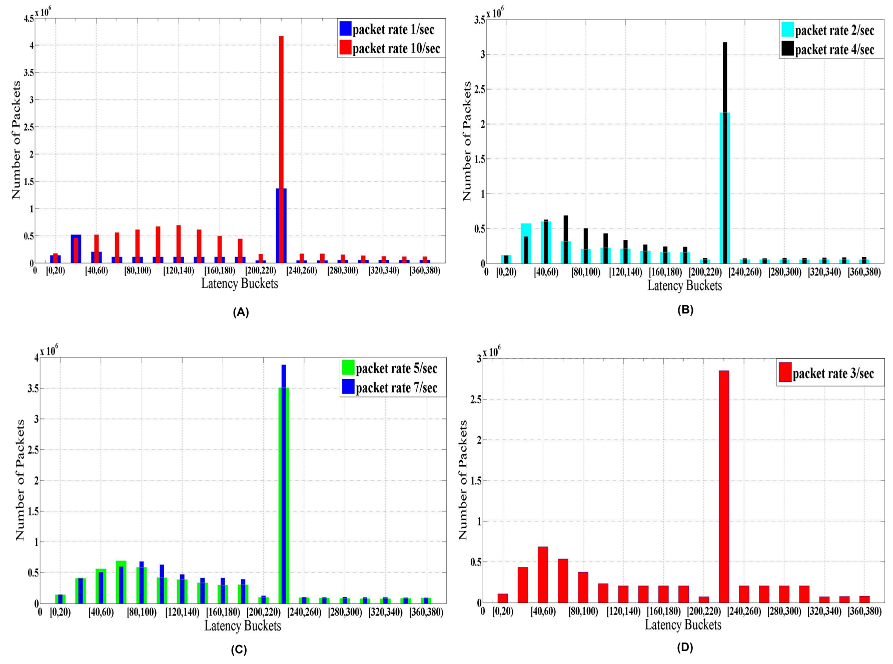

Fig. 17 Delay incurred at the base station

The simulation tool employed allows us to consider other measurements in addition to the life time of the WSN nodes. Various performance measures such as response time (latency), the number of dropped packets etc. can be analysed in details. Figure 17 shows the latency of the packets received by the monitoring station located in the living room. The results show that most of the packets have a latency interval of 230 ms. As the packet rate increases, the number of packets within the same latency interval also increases. The figure also demonstrates the flexibility of the considered system in terms of performance, availability, and energy-related measures.

8 Related Work

In order to simplify the design and configuration of the WSN at large, and abstract from technical low-level details, a number of Model-Driven Engineering (MDE) approaches or of modeling notations for WSN engineering have been proposed. Those approaches are used to specify a WSN at different levels of abstraction (hardware, application, communication protocols, etc.) with the recurrent goals of code generation, communication overhead analysis, and energy consumption.

The following of this section is structured so to cover three related research areas: i) frameworks for the engineering of WSNs (or related domains), ii) domain-specific modelling languages for WSN, iii) other modeling and analysis approaches for engineering WSNs, and iv) surveys related to the modeling and analysis of WSNs.

Frameworks for Engineering WSNs (or related domains): Engineering frameworks strongly related to A4WSN have been presented in [8,61,50].

In Reference [8] the authors propose DiaSuite, a tool suite proposed for the design, analysis, and deployment of Sense/Compute/Control applications. The DiaSuite domain specific design language supports the modeling of a taxonomy layer and an application design layer. Those models are successively used to generate a dedicated Java programming framework (to guide and support programmers to implement the various parts of the software system), for simulation purposes, and for deploying the application on a specific execution platform. When compared with A4WSN, DiaSuite has similar goals (modeling for analysis a code generation), but covers a different domain. As a result, the modeling languages are extremely different, and are manipulated (for analysis and code generation) in different ways.

In Reference [61] a set of modelling languages is the starting point for code generation and performance (with energy consumption) analysis. Those languages are based on concepts such as sampling task, aggregation task, network communication tasks, etc. and they are the starting point of a model-driven process to enable a low-cost prototyping and optimisation of WSN applications. In [46], a framework for modelling, simulation, and code generation of WSNs is presented. The framework is based on Simulink, Stateflow and Embedded Coder, and allows engineers to simulate and automatically generate code with energy as one of the main issues.

In Reference [50] a multi-stage model-driven approach for IoT applications development has been proposed. Such an approach takes into explicit consideration the existence of five different types of IoT stakeholders, and according to their needs, propose five different modelling languages. Those models are successively used for code-generation and task-mapping techniques. Similarly to DiaSuite [8], while sharing the same goals of A4WSN, the framework in [50] covers a different (still, related) domain.

Domain-specific modelling languages for WSN: many approaches propose to use Domain-specific Modelling Languages (DSML) for representing WSNs from different viewpoints. For example, in [67] the proposed modelling language contains concepts such as node group, region, resource, wireless link; whereas, in [19] authors propose a set of languages spanning from application-level actions (e.g., sense, send message, store data) to hardware specifications (e.g., processor, sensing devices, radio tran- sceivers), and so on. In [65] the authors propose Verisensor, a DSML based on concepts such as system, node class, application etc., with the possibility to automatically translate models towards a formal language for checking the life time of the WSN and its correct behaviour.

In Reference [15], the authors propose the LWiSSy domain specific language for wireless sensor and actuator network systems. The LWiSSy metamodel comprises three views: structural behavioral, and optimization. Those three views are described in details, and successively evaluated through a controlled experiment.

Other approaches, such as those proposed in [45] and [21], are based on *generic modelling languages*. They mainly use extensions of UML and Simulink for representing a WSN.

In order to better understand how MDE has been used for designing and analysing wireless sensor networks, [39] surveys and classifies state-of-the-art MDE approaches for engineering WSNs.

Other modelling and analysis approaches for engineering WSNs: describing a network from a *structural* point of view is very straightforward and easy to reason on (just think about the component-based representation in Omnet++[16], one of the most popular network simulators). Also, an approach based on DDS (i.e., the data-centric middleware standard introduced by OMG) is presented in [6]; the authors proposed four types of modelling languages (namely for data types, data space, node structure, and node conguration) and use them as input for a set of optimisation and transformation steps, eventually delivering deployable application code as output.

Also, in some cases (e.g., when capabilities such as fault tolerance and security analysis are needed) the structure of WSNs may not be enough, and thus describing the *behaviour* of the WSN is fundamental. In [25], the authors address energy-aware system design of Wireless Sensor Networks (WSNs). Energy mode signalling and energy scheduling of nodes within a WSN are represented as SDL models, and then analysed.

Rodrigues et al. in [54] proposes an MDA process where application domain experts model the Platform Independent Model (PIM) of a WSN application. Such a PIM is successively transformed into a Platform Specific Model (PSM) and refined by a network expert. Class and Activity diagrams are used to specify the WSN application at the PIM level, while Component and Finite State diagrams are used at the PSM level.

An approach for formal modeling and analysis of WSN in Real-Time Maude is presented in [48]. In [59] Samper et al. propose the GLONEMO formal model for the analysis of ad-hoc sensor networks.

For what concerns the *physical environment* of a WSN, the majority of approaches in the literature does not allow designers to specify the *physical deployment* of the WSN nodes. Among those that support (in some form) this feature, there is great variability. There are some approach which support an explicit definition of the physical environment (e.g., in [19] the tool allows engineers to model real-world dimensions, obstacles with attenuation coefficients, etc.); others allow designers to define physical quantities (e.g., in [7] engineers can define models of the evolution of each physical quantity in a given scenario), and so on. However, all these approaches do not provide any intuitive and abstract means to easily define the deployment environment of the WSN. A recent study [40] has investigated how WSN engineers currently specify the physical environment and how they would like to do it.

Surveys related to the modeling and analysis of WSNs: A survey on system models in WSNs has been conducted in [64]: there the authors identify several dimensions

[16] http://www.omnetpp.org/

to be used to classify model (types) used to specify networked computing systems (from models of signal propagation, to models of the application). Existing models are then organized into a taxonomy. In Reference [34] the authors survey 9 WSN modeling techniques. Through this study, they show how each technique models different parts of the system. The models here analyzed are extensions to existing notations, such as SDL, Promela, UML, and others.

Final Remarks: A4WSN shares with some of the related approaches above the wish to provide a *clear separation of concerns* between different modeling views, to enhance *reuse*, to *abstract* from low level details, and to support *early analysis* of WSN applications. What distinguishes A4WSN from other related work are i) the modeling languages that have been selected for modeling WSN applications, including an explicit graphical modeling of the application physical environment, ii) the definition of models dedicated to the weaving of the three main modeling languages, iii) the existence of an extensible programming framework that enables third-party researchers and developers to reuse the A4WSN modelling environment and programming framework when developing new analysis and code generation engines. In this context, third-party researchers can focus exclusively on solving their peculiar issues, while spending minimal effort and implementation time on realizing the facilities already provided by A4WSN out of the box. iv) The maturity of A4WSN with respect to other approaches that, while sharing some of our desires, seem to still implement only a subset of them.

9 Future Work and Final Remarks

In this paper we proposed a modelling platform supported by a dedicated programming framework for the model-driven engineering of wireless sensor networks. The modelling viewpoints and conceptual elements have been carefully designed in collaboration with colleagues from various domains, such as software engineering, wireless sensor networks, and telecommunications. The programming framework functioning has been tested by realizing a plugin devoted to energy-related simulation of WSNs.

The modeling and programming framework represent only the starting point of a series of goals we are willing to achieve in the mid-term.

Firstly, we plan to have the framework used by practitioners involved in the development of WSNs. We wish to record and analyze their usage patterns and collect their feedback for further improving our platform.

Secondly, we are aware that it might be necessary to extend the modelling languages to provide additional concepts for supporting new analysis or code generation engines. For example, we are working on providing new SAML data structures (either primitive or structured), new attributes for better specifying nodes in the NODEML modelling language, and on the extension of the purposefully simple ENVML modelling language (e.g., by adding multi-floor support for indoor deployments, by supporting the specification of properties specific to outdoor setups, etc.). In this context, introducing changes at the metamodel level might have a strong impact on the already developed plugins (model editors, model transformations, etc.). This problem is called metamodel co-evolution management and it is well-known in the MDE research field [12, 56]. If we look at this problem from a different perspective, similarly to what we proposed in a previous work on architectural languages interoperability [58], a possible solution could be to provide a systematically defined extension process for our modelling languages. According to this extension process, languages extensions are organized into a hierarchy obtained by systematically extending a root modelling language. Under this perspective, we plan to build on (and adapt, if needed) metamodel co-evolution techniques [57, 32] in order to tackle this problem.

Thirdly, we would like to realize an analysis plug-in that, while getting in input a series of environmental configurations options, can tell us which configuration can increase the network life-time (so far, PlaceLife can evaluate the expected life-time of a given configuration, but is quite impractical for comparatively analyse alternative solutions). We plan to use genetic algorithms and search-based approaches to achieve such a goal.

Finally, we are working on a WSN performance analysis plugin that allows engineers to run a trade-off analysis between energy consumption and performance indices like sensor nodes throughput, reliability and network latency.

Acknowledgments

The authors of this paper would like to thank Dajana Cassioli, Vittorio Cortellessa, and Luigi Pomante for their comments and suggestions on the modelling environments. This research has been partially supported by the Ricostruire R.I.D.I.T.T. (Rete Italiana per la Diffusione dell′ Innovazione e il Trasferimento Tecnologico alle imprese) project.

References

1. (2012). URL {http://www.mathopenref.com/coordbounds.html}. Definition of minimum bounding box
2. (2012). URL {http://usa.autodesk.com/autocad/}. AutoCAD website
3. Al-karaki, J.N., Kamal, A.E.: Routing techniques in wireless sensor networks: A survey. IEEE Wireless Communications **11**, 6–28 (2004)
4. Alemdar, H., Ersoy, C.: Wireless sensor networks for healthcare: A survey. Comput. Netw. **54**(15), 2688–2710 (2010). DOI 10.1016/j.comnet.2010.05.003
5. Alwan, M., Leachtenauer, J., Dalal, S., Mack, D., Kell, S., Turner, B.: Impact of monitoring technology in assisted living: Outcome pilot. IEEE Transactions on Information Technology in Biomedicine **10** (2006)
6. Beckmann, K., Thoss, M.: A model-driven software development approach using OMG DDS for wireless sensor networks. In: Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems, SEUS'10, pp. 95–106. Springer-Verlag, Berlin, Heidelberg (2010)
7. Ben Maïssa, Y., Kordon, F., Mouline, S., Thierry-Mieg, Y.: Modeling and Analyzing Wireless Sensor Networks with VeriSensor. In: Petri Net and Software Engineering (PNSE), vol. 851, pp. 60–76. CEUR, Hamburg, Germany (2012)
8. Benjamin Bertran and Julien Bruneau and Damien Cassou and Nicolas Loriant and Emilie Balland and Charles Consel: DiaSuite: A tool suite to develop Sense/Compute/Control applications . Science of Computer Programming **79**(0), 39 – 51 (2014). Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010)

9. Bjornemo, E., Johansson, M., Ahlen, A.: Two hops is one too many in an energylimited wireless sensor network. In: in Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, pp. 181–184 (2007)
10. Blumenthal, J., Handy, M., Golatowski, F., Haase, M., Timmermann, D.: Wireless sensor networks - new challenges in software engineering. In: Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA '03. IEEE Conference, vol. 1, pp. 551 – 556 vol.1 (2003)
11. C. Szyperski: Component Software. Beyond Object Oriented Programming. Addison Wesley (1998)
12. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: 12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, 15-19 September 2008, Munich, Germany, pp. 222–231. IEEE Computer Society (2008)
13. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE, pp. 422–437 (2005)
14. van Dam, T., Langendoen, K.: An adaptive energy-efficient mac protocol for wireless sensor networks. In: Proceedings of the 1st international conference on Embedded networked sensor systems, SenSys '03, pp. 171–180. New York, NY, USA (2003)
15. Dantas, P., Rodrigues, T., Batista, T., Delicato, F., Pires, P., Li, W., Zomaya, A.: Lwissy: A domain specific language to model wireless sensor and actuators network systems. In: Software Engineering for Sensor Network Applications (SESENA), 2013 4th International Workshop on, pp. 7–12 (2013). DOI 10.1109/SESENA.2013.6612258
16. Del Fabro, M.D., Valduriez, P.: Semi-automatic model integration using matching transformations and weaving models. In: Proceedings of the 2007 ACM Symposium on Applied Computing, SAC '07, pp. 963–970. ACM, New York, NY, USA (2007). DOI 10.1145/1244002.1244215. URL http://doi.acm.org/10.1145/1244002.1244215
17. Demirkol, I., Ersoy, C., Alagoz, F.: MAC protocols for wireless sensor networks: a survey. IEEE Communications Magazine **44**(4), 115–121 (2006). DOI 10.1109/mcom.2006.1632658
18. Didonet Del Fabro M., Bézivin J., Jouault F. and Breton E. and Gueltas G.: AMW: a generic model weaver. In: Proc. of 1re Journe sur l'Ingnierie Dirige par les Modles, Paris, France. pp 105-114 (2005)
19. Doddapaneni, K., Ever, E., Gemikonakli, O., Malavolta, I., Mostarda, L., Muccini, H.: A model-driven engineering framework for architecting and analysing wireless sensor networks. In: SESENA, pp. 1–7 (2012)
20. Evans, D.: The Internet of Things - How the Next Evolution of the Internet is Changing Everything (2011). CISCO White Paper
21. Fuchs, G., German, R.: Uml2 activity diagram based programming of wireless sensor networks. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications, SESENA '10, pp. 8–13. ACM, New York, NY, USA (2010). DOI 10.1145/1809111.1809116
22. Gill, K., Yang, S.H., Yao, F., Lu, X.: A zigbee-based home automation system. Consumer Electronics, IEEE Transactions on **55**(2), 422 –430 (2009)
23. Gluhak, A., Krco, S., Nati, M., Pfisterer, D., Mitton, N., Razafindralambo, T.: A survey on facilities for experimental internet of things research. Communications Magazine, IEEE **49**(11), 58–67 (2011). DOI 10.1109/MCOM.2011.6069710
24. Goldsmith, A.: Wireless Communications. Cambridge University Press, New York, NY, USA (2005)
25. Gotzhein, R., Krämer, M., Litz, L., Chamaken, A.: Energy-aware system design with SDL. In: Proceedings of the 14th international SDL conference on Design for motes and mobiles, SDL'09, pp. 19–33. Springer-Verlag, Berlin, Heidelberg (2009)
26. Han, D.M., Lim, J.H.: Smart home energy management system using ieee 802.15.4 and zigbee. Consumer Electronics, IEEE Transactions on **56**(3), 1403 –1410 (2010)
27. Heinzelman, W.R., Chandrakasan, A., Balakrishnan, H.: Energy-efficient communication protocol for wireless microsensor networks. In: Proc. of the 33rd Hawaii Intl Conf. on System Sciences (HICSS). Washington, DC, USA (2000)
28. Hill, J.L.: System architecture for wireless sensor networks. Ph.D. thesis, University of California, Berkeley (2003). AAI3105239
29. Huang, C.F., Tseng, Y.C.: The coverage problem in a wireless sensor network. In: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications, WSNA '03, pp. 115–121. ACM, New York, NY, USA (2003). DOI 10.1145/941350.941367
30. Imran, M., Said, A., Hasbullah, H.: A survey of simulators, emulators and testbeds for wireless sensor networks. In: Information Technology (ITSim), 2010 International Symposium in, vol. 2, pp. 897–902 (2010). DOI 10.1109/ITSIM.2010.5561571

31. Intanagonwiwat, C., Estrin, D., Govindan, R., Heidemann, J.: Impact of network density on data aggregation in wireless sensor networks. In: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02), ICDCS '02, pp. 457–. IEEE Computer Society, Washington, DC, USA (2002)
32. Iovino, L., Pierantonio, A., Malavolta, I.: On the impact significance of metamodel evolution in mde. Journal of Object Technology **11**(3), 3: 1–33 (2012)
33. ISO/IEC/IEEE: ISO/IEC/IEEE 42010:2011 Systems and software engineering – Architecture description (2011)
34. Khalil, J., Liscano, J.R., Bradbury, J.: A survey of modeling techniques for wireless sensor networks. In: SENSORCOMM 2011, The Fifth International Conference on Sensor Technologies and Applications, pp. 103–109 (2011)
35. K.Pahlavan and P.Krishnamurthy: Networking Fundamentals. John Wiley and Sons, Chichester, UK (2009)
36. Lorincz, K., Malan, D., Fulford-Jones, T., Nawoj, A., Clavel, A., Shnayder, V., Mainland, G., Welsh, M., Moulton, S.: Sensor networks for emergency response: challenges and opportunities. Pervasive Computing, IEEE **3**(4), 16 – 23 (2004). DOI 10.1109/MPRV.2004.18
37. Losilla, F., Vicente-Chicote, C., lvarez, B., Iborra, A., Snchez, P.: Wireless Sensor Network Application Development: An Architecture-Centric MDE Approach. In: F. Oquendo (ed.) ECSA, *LNCS*, vol. 4758, pp. 179–194. Springer (2007)
38. malavolta, I., Mostarda, L., Muccini, H., Doddapaneni, K.: The A4WSN Modelling languages (2013). URL http://a4wsn.di.univaq.it/files/a4wsnLanguages.pdf
39. Malavolta, I., Muccini, H.: A study on mde approaches for engineering wireless sensor networks. In: In Proc. of the 40th Euromicro Conference series on Software Engineering and Advanced Applications (SEAA), August 2014 (2014)
40. Malavolta, I., Muccini, H.: A survey on the specification of the physical environment of wireless sensor networks. In: In Proc. of the 40th Euromicro Conference series on Software Engineering and Advanced Applications (SEAA), August 2014 (2014)
41. Malavolta, I., Muccini, H., Pelliccione, P., Tamburri, D.: Providing architectural languages and tools interoperability through model transformation technologies. Software Engineering, IEEE Transactions on **36**(1), 119 –140 (2010)
42. Mottola, L., Pathak, A., Bakshi, A., Prasanna, V., Picco, G.: Enabling scope-based interactions in sensor network macroprogramming. In: Mobile Adhoc and Sensor Systems, 2007. MASS 2007. IEEE Internatonal Conference on, pp. 1–9 (2007). DOI 10.1109/MOBHOC.2007.4428655
43. Mottola, L., Picco, G.P.: Programming wireless sensor networks: Fundamental concepts and state of the art. ACM Comput. Surv. **43**, 19:1–19:51 (2011)
44. Mottola, L., Picco, G.P.: Middleware for wireless sensor networks: an outlook. J. Internet Services and Applications **3**(1), 31–39 (2012)
45. Mozumdar, M., Gregoretti, F., Lavagno, L., Vanzago, L., Olivieri, S.: A framework for modeling, simulation and automatic code generation of sensor network application. In: Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON '08. 5th Annual IEEE Communications Society Conference on, pp. 515 –522 (2008). DOI 10.1109/SAHCN.2008.68
46. Mozumdar, M.M.R., Gregoretti, F., Lavagno, L., Vanzago, L., Olivieri, S.: A framework for modeling, simulation and automatic code generation of sensor network application. In: SECON, pp. 515–522 (2008)
47. Newport, C., Kotz, D., Yuan, Y., Gray, R.S., Liu, J., Elliott, C.: Experimental evaluation of wireless simulation assumptions. Simulation **83**(9), 643–661 (2007). DOI 10.1177/0037549707085632
48. Olveczky, P., Thorvaldsen, S.: Formal modeling and analysis of wireless sensor network algorithms in real-time maude. In: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, pp. 8 pp.– (2006). DOI 10.1109/IPDPS.2006.1639414
49. Pahlavan, K., Krishnamurthy, P.: Networking Fundamentals: Wide, Local and Personal Area Communications. John Wiley & Sons (2009)
50. Patel, P., Pathak, A., Cassou, D., Issarny, V.: Enabling High-Level Application Development in the Internet of Things. In: M. Zuniga, G. Dini (eds.) Sensor Systems and Software, *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, vol. 122, pp. 111–126. Springer International Publishing (2013). URL http://dx.doi.org/10.1007/978-3-319-04166-7_8
51. Picco, G.P.: Software engineering and wireless sensor networks: happy marriage or consensual divorce? In: Proceedings of the FSE/SDP workshop on Future of software engineering research, FoSER. NY, USA (2010)

52. Rajagopalan, R., Varshney, P.: Data-aggregation techniques in sensor networks: a survey. Communications Surveys Tutorials, IEEE **8**(4), 48–63 (2006). DOI 10.1109/COMST.2006.283821
53. Rappaport, T.: Wireless communications: principles and practice. Prentice Hall communications engineering and emerging technologies series. Prentice Hall PTR (1996)
54. Rodrigues, T., Batista, T., Delicato, F., Pires, P., Zomaya, A.: Model-driven approach for building efficient wireless sensor and actuator network applications. In: Software Engineering for Sensor Network Applications (SESENA), 2013 4th International Workshop on, pp. 43–48 (2013). DOI 10.1109/SESENA.2013.6612265
55. Romer, K., Mattern, F.: The design space of wireless sensor networks. Wireless Communications, IEEE **11**(6), 54 – 61 (2004)
56. Rose, L., Etien, A., Méndez, D., Kolovos, D., Paige, R., Polack, F.: Comparing model-metamodel and transformation-metamodel coevolution. In: International Workshop on Models and Evolutions (2010)
57. Ruscio, D.D., Iovino, L., Pierantonio, A.: Coupled evolution in model-driven engineering. IEEE Software **29**(6), 78–84 (2012)
58. Ruscio, D.D., Malavolta, I., Muccini, H., Pelliccione, P., Pierantonio, A.: Model-driven techniques to enhance architectural languages interoperability. In: FASE, pp. 26–42 (2012)
59. Samper, L., Maraninchi, F., Mounier, L., Mandel, L.: Glonemo: Global and accurate formal models for the analysis of ad-hoc sensor networks. In: Proceedings of the First International Conference on Integrated Internet Ad Hoc and Sensor Networks, InterSense '06. ACM, New York, NY, USA (2006). DOI 10.1145/1142680.1142684. URL http://doi.acm.org/10.1145/1142680.1142684
60. Seybold, John S: Introduction to RF Propagation. Wiley, Newark, NJ (2005)
61. Shimizu, R., Tei, K., Fukazawa, Y., Honiden, S.: Model driven development for rapid prototyping and optimization of wireless sensor network applications. In: Proceedings of SESENA '11, pp. 31–36. ACM, New York, NY, USA (2011)
62. Shnayder, V., Chen, B.r., Lorincz, K., Jones, T.R.F.F., Welsh, M.: Sensor networks for medical care. In: Proceedings of the 3rd international conference on Embedded networked sensor systems, SenSys '05, pp. 314–314. ACM, New York, NY, USA (2005). DOI 10.1145/1098918.1098979
63. Stankovic, J.A.: Research challenges for wireless sensor networks. SIGBED Rev. **1**, 9–12 (2004)
64. Stanley-Marbell, P., Basten, T., Rousselot, J., Oliver, R.S., Karl, H., Geilen, M., Hoes, R., Fohler, G., Decotignie, J.D.: System models in wireless sensor networks. Tech. Rep. ESR-2008-06, Eindhoven University of Technology (2008)
65. Thang, N.X., Geihs, K.: Model-driven development with optimization of non-functional constraints in sensor network. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications, SESENA '10, pp. 61–65. ACM, New York, NY, USA (2010). DOI 10.1145/1809111.1809128
66. Varga, A., Hornig, R.: An overview of the omnet++ simulation environment. In: Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops, pp. 1–10 (2008)
67. Vicente-Chicote, C., Losilla, F., Álvarez, B., Iborra, A., Sánchez, P.: Applying MDE to the Development of Flexible and Reusable Wireless Sensor Networks. Int. J. Cooperative Inf. Syst. **16**(3/4), 393–412 (2007)
68. Willig, A.: Wireless sensor networks: concept, challenges and approaches. Elektrotechnik & Informationstechnik **123**(6), 224 –231 (2006)
69. Yang, G.Z., Yacoub, M.: Body sensor networks, vol. 6. Springer London (2006)
70. Ye, W., Heidemann, J., Estrin, D.: Medium access control with coordinated adaptive sleeping for wireless sensor networks. IEEE/ACM Trans. Netw. **12**(3), 493–506 (2004). DOI 10.1109/TNET.2004.828953