The A4WSN Modelling languages

Ivano malavolta

i.malavolta@vu.nl

Abstract

This document describes the modelling languages introduced for A4WSN: an architecture-based modelling framework that supports development and analysis of Wireless Sensor Networks (WSNs). The A4WSN framework uses different models to specify different concerns such as structure, behaviour, hardware and physical environment. Models are linked together in order to perform analysis and code generation. The framework supports the addition of new plug-ins in order to provide additional code generation and new analysis engines (this part of the framework is not described in this document though).

Keywords: WSN, Software Architecture, MDE

1. SAML Metamodel

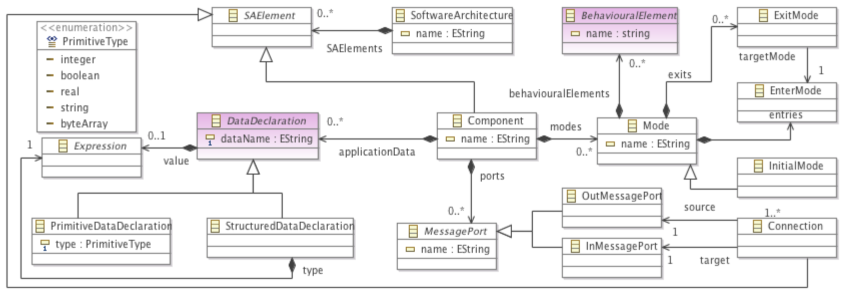

Figure 1: SAML Metamodel: structural part

SoftwareArchitecture. A collection of software *components* and *connections* instantiated in a configuration. It represents the application layer of the WSN and it is the root element of every SAML model.

SAElement. The superclass for all the elements that can be contained into a software architecture (namely, *components* and *connections*).
Component. A unit of computation with internal state and well-defined interface [1]. Components interact with other components in the same software architecture through *message ports*. A component can contain a set of *application data*, which are used to model the internal state of the components. A component can also contain a set of behavioural modes (see the *modes* containment reference linking the Component metaclass in Figure 1 and the *Mode* metaclass in Figure 2); each mode can contain the various actions, conditions and events describing the behaviour of the component.
Connection. A unidirectional communication channel between two message ports of two different components. Communication happens by message passing, that is, a component can send messages from one of its output message ports to input ports of other 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).
MessagePort. It specifies the interaction point between a component and its external environment. These are the loci in which messages can either depart (arrive) to (from) components. Depending on the application being modelled, each *Component* can have an arbitrary number of message ports.
InMessagePort. A specialization of *MessagePort* representing input message ports, i.e., ports where incoming messages can be received.
OutMessagePort. A specialization of *MessagePort* representing output message ports, i.e., ports where outgoing messages can be sent.
DataDeclaration. It represents a local variable declared in the scope of the component containing it. Data declarations are manipulated by actions, events, and conditions defined in the behaviour of the component, such as *StoreData*, *ReceiveMessage*, etc. Depending on the application being modelled, each *Component* can have an arbitrary number of data declarations.
PrimitiveDataDeclaration. It represents a data declaration which may have only a primitive type. The primitive types available in 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. The syntax for creating a new primitive data declaration is the following: $< dataName >:< type >=< value >$, where *dataName*, *type*, and *value* refer to the homonymous attributes and references in the SAML metamodel in Figure 1.

Listing 1 shows some examples of primitive data declarations.

```
threshold : integer = 30
temperature : real = 15.5
message : string = "hello"
found : boolean = false
imageStream : bytearray = "ABCD" // byte array constants are represented as
    strings
```

Work in progress. The byarray data type is not supported in the current version of the A4WSN tool.

Listing 1: Examples of primitive data declaration.

StructuredDataDeclaration[1]. It represents a data declaration which may have a type composed of other (usually primitive) types. Also, differently from primitive types (which may have only a single primitive value), structured types can have user-defined values. A structured data type can be either an *Enumeration*, a *Structure*, an *Array*, or a *Map*[2]. An *enumeration* data type represents a user-defined data type. An enumeration contains a set of identifiers that represent the only possible values of the enumeration. A *structure* data type represents a user-defined data type which can contain an arbitrary set of other data types. The set of internal data types of an SAML structure can contain either primitive or structured data types and it can be defined only when the structure data type is declared. In SAML a structure data type is similar to a Java class in which Java attributes correspond to the internal data types of the SAML structure. An *array* data type represents a collection of elements of the same type, which can be retrieved by an integer index representing their position within the collection. Those elements can be either primitive or structured. In SAML the type of the elements is defined when the array is declared and cannot be changed. SAML provides predefined expression types for (i) accessing an element of the array at a given index (if the provided index is outside the boundaries of the array, SAML returns a null value), and (ii) to get the number of the elements which are currently stored in the array. A *map* data type represents an associative array, a very well-known data structure in computer science. Basically, a map contains a set of key-value pairs. A value can be lookup and accessed by referring to its associated key, which must be unique within the map. In SAML, a key is a string, and values can be either primitive or structured. SAML provides predefined expressions for getting and setting an element of a map by its key. When a key is not present within the map, SAML returns a null value. Listing 2 shows some examples of structured data

Work in progress. The array, bytearray, and map data types are not supported in the current version of the A4WSN tool.

[1]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

[2]Arrays and maps are not supported in the current version of the A4WSN tool.

declarations.

```
status : enum{ON, OFF} = status.OFF
gender : enum{MALE, FEMALE}
person : struct{name: string, sex: gender} = person("John", gender.MALE)
readings : array{integer} = [12, 34, 56]
rooms : map = {"key1": 5, "key2": status.OFF}
```

Work in progress. The array and map data types are not supported in the current version of the A4WSN tool.

Listing 2: Examples of structured data declaration.

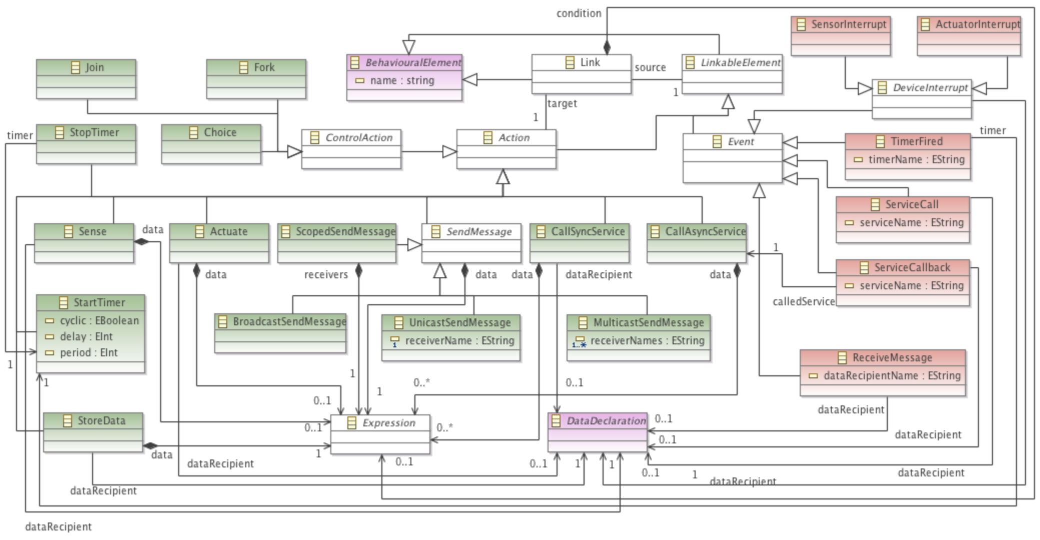

Figure 2: SAML Metamodel: behavioural part

BehaviouralElement. The root of the metamodel, i.e., every metaclass (but *Mode*) specializes *BehaviouralElement* either directly or indirectly. Every BehaviouralElement has a name.

LinkableElement. It is an abstract class extending *BehaviouralElement*, it represents every element that can be linked in the behavioural description of a component, such as events and actions.

Action. A special kind of behavioural element which represent an action 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.

Event. A special kind of behavioural element which represent an event that can be triggered 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 message port), or some internal mechanism of the component (e.g., a timer fired).

Link. It represents the control flow among events and actions. Fundamentally, it helps architects in (i) defining the order in which actions can be executed, and (ii) which actions must be executed when a given 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 description of the *Expression* metaclass at the end of this section for further details).
Mode. A mode is a specific status of the component. Examples of modes can be: sleeping mode, energy saving mode, etc. It is important to note that modes are defined at the application layer in SAML, thus they can, but are not forced to, be related to the energy-related modes of the WSN node they are running on. 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. *Mode* is the only metaclass that is not included in the specialization sub-tree of *BehaviouralElement*; we made this choice for preventing architects to define nested modes in SAML models, thus keeping the SAML language simple to some extent. A component can switch from a mode to another by means of the previously defined *Link* construct. 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.
EnterMode. A specialization of the *Event* metaclass; it represents a special kind of event which is triggered when the behavioural flow enters a specific mode. Actions can be linked to this type of events, they will be executed immediately after the entered mode becomes active.
ExitMode. A specialization of the *Action* metaclass; it represents the action of exiting a specific mode, and thus entering another mode within the component containing them (see the *targetMode* reference of *ExitMode* in Figure 2.
InitialMode. A special kind of mode. Intuitively, an initial mode is the first mode which is active when the component starts up. Clearly, each component can contain one and only one initial mode.
Sense. An action that gets some data from a sensor and stores the read value into a specific application data declaration. This action can be used, for example, to

model the retrieval of the current temperature in the environment, to sense for the presence of smoke, etc.

Actuate. An action that activates an actuator; optionally, an expression can be used to pass a parameter to the actuator. Also, an actuate action can optionally refer to an application data declaration to which the result of the actuate action can be stored. This action can be used, for example, to model the action of turning off a light or activating the heating system of a house.

SendMessage. An abstract metaclass representing the action of sending a message via a specific output message port (for the sake of clarity we did not show the *outMessagePort* reference from *SendMessage* to *OutMessagePort* in Figure 2). Optionally, the contents of the message can be specified by passing an expression as parameter. Available types of message include: broadcast message, multicast message, and unicast message; they differ in the number of target physical nodes of the WSN actually receiving the message. By target physical node we mean all the nodes containing the components declaring an *InMessagePort* that is connected to the *OutMessagePort* referenced by the send message action.

BroadcastSendMessage. A special kind of *SendMessage* action in which the message is sent to every node containing the target component.

MulticastSendMessage. A special kind of *SendMessage* action in which the message is sent to a specific set of nodes containing the target component. Those nodes are represented by a list of receivers, identified by the name they will have in the final deployment of the WSN.

UnicastSendMessage. A special kind of *SendMessage* action in which the message is sent to a single node containing the target component. The receiver node is identified by its name in the final deployment of the WSN.

ScopedSendMessage. A special kind of *SendMessage* action in which the message is sent to a specific set of nodes containing the target component. The set of nodes receiving the message is computed at run-time, depending on the value of the *receivers* boolean expression; basically, each node whose application data values let the *receivers* expression to evaluate to $true$ will receive the specific message. 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.

Work in progress. The ScopedSendMEssage action is not supported in the current version of the A4WSN tool.

StoreData. An action that puts some data into an application data declaration of the component. In order to allow architects to manipulate data when storing it, the data to be stored is defined as an *Expression*.

CallSyncService. An action that calls an external service (e.g., a web service, another entity which is not part of the WSN, etc.). As suggested by the name of

the action, the called external service is synchronous, i.e., the behavioural flow is stopped until the result of the service call is available; such a result is stored into the application data declaration of the component specified in the *dataRecipient* reference. Architects can also specify a set of parameters that can be passed to the called service; they are specified in the *data* reference of type *Expression*; in order to support the definition of multiple parameters, this reference has cardinality "0..*".
CallAsyncService. This action is similar to *CallSyncService*, the only difference is that the called service is asynchronous, rather than a synchronous one. When the result of the called service is available, a special event, named *ServiceCallback*, is triggered in the component.
StartTimer. An action that starts an internal timer in the application. The expiration of a started timer is represented by a special event called *TimerFired*. When starting a timer, architects can set three parameters: (i) the *delay* (in milliseconds) that must occur before the first activation of the timer, (ii) the *cyclic* nature of the timer (that is, whether it must be periodic or not), and (iii) the *period* of the timer (in milliseconds), if it is a cyclic one.
StopTimer. This action stops a previously started *timer*.
ControlAction. An abstract action representing any action that is used to sync and split the control flow.
Fork. A specialization of *ControlAction* representing the classical *fork* operation on a (control) flow graph. Basically, this action is used to explicitly split the incoming behavioural flow into a set of parallel flows.
Join. A specialization of *ControlAction* representing the classical *join* operation on a (control) flow graph. Intuitively, it performs the inverse operation of a fork operation, i.e., it merges incoming behavioural flows and syncs them into a common outgoing one.
Choice. A specialization of *ControlAction* representing the classical *choice* operation on a (control) flow graph. This action is used to split the control flow into one or more branches. Depending on the value of the conditions in the outgoing behavioural links, one and only one control flow is executed after the choice control action.
ReceiveMessage. An event triggered when the component receives a message to one of its input message ports (referenced by *fromMessagePorts*, not shown in Figure 2 for the sake of readability). The *dataRecipient* reference points to the application data declaration which will hold the contents of the received message.
ServiceCallback. An event that is triggered when the results of some previously called asynchronous service (referenced by *calledService*) is available to the com-

ponent. The result of the called service is available in the application data declaration defined in *dataRecipient*. The service providing the result must have been called in the same component in which the corresponding *ServiceCallback* event is used.

ServiceCall. An event triggered when some kind of external service interacts with the component. Examples of the occurrence of this event include: a web server pushes data to a node of the WSN, a software entity outside the WSN pro-actively interacts with a node of the WSN, etc. The *dataRecipient* reference points to the application data declaration which will hold the actual parameters of the service call.

TimerFired. An event triggered every time a previously started timer expires. The previously started timer is defined by the *timer* reference in the *FiredTimer* metaclass.

DeviceInterrupt. An event triggered every time a specific device (which can be either a sensor or an actuator) proactively provides some information about the environment. For example, this kind of event can be used to represent the activation of a movement sensor, or some specific sensor which continuously monitors the environment until a specific situation occurs.

Work in progress. The Device interrupt (and thus also all its subtypes) are not supported in the current version of the A4WSN tool.

SensorInterrupt. A special kind of *DeviceInterrupt* in which the device triggering the interrupt is a sensor.

ActuatorInterrupt. A special kind of *DeviceInterrupt* in which the device triggering the interrupt is an actuator.

Expression. This metaclass represents any kind of operation which is related to an application data. SAML exposes four types of expression to architects:

1. *Constant*: a constant representing the value of some application data, like an integer, a real, an instance of structure, etc.;
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 Java class;
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;

Work in progress. String operations are not supported in the current version of the A4WSN tool.

Since the *Expression* metaclass described classical elements in most programming languages, we do not show the portion of metamodel related to it.

2. NODEML Metamodel

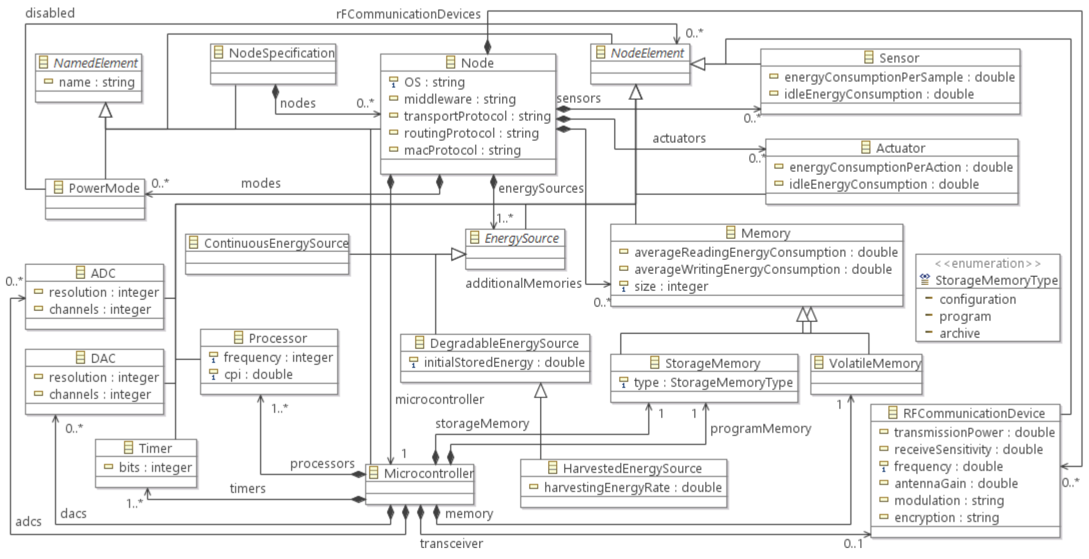

Figure 3: NODEML Metamodel

NamedElement. Each element within a NODEML model is a specialization of this metaclass. Basically, this means that each element of a NODEML model can have a *name*.
NodeSpecification. It is the root element of a NODEML model. It is a container of instances of the *Node* metaclass (described below).
Node. It represents a specific node type that can be used within the WSN. A node instance can have a name (inherited from the NamedElement metaclass), and designers must specify its Operating System (i.e., its *OS* such as TinyOS [2], Contiki [3], Mantis [4], LiteOS [5]), the *macProtocol* (such as T-MAC, S-MAC, WiseMAC, SIFT, ecc. [6]) and *routingProtocol* (such as SPIN, LEACH, GEAR, ecc. [7]) it uses to communicate with other nodes within the WSN.

According to NODEML, a WSN node contains the following elements: one or more energy sources (e.g., batteries), a micro-controller (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.

NodeElement. An abstract metaclass representing every element which can be contained into a node definition (i.e., an instance of the *Node* metaclass).

EnergySource. It represents any energy source of the node. *EnergySource* is an abstract metaclass, and it is extended by the following three metaclasses, representing continuous, degradable, and harvested energy sources, respectively.

ContinuosEnergySource. A specialization of *EnergySource* which potentially never runs out. A classical example of this kind of energy source is AC/DC supply.

DegradableEnergySource. A specialization of *EnergySource* representing energy sources that can terminate at any time. The *initialStoredEnergy* attribute describes the amount of energy (in Joule) initially stored in the energy source. As an example, batteries can be represented by a degradable energy source in NODEML.

HarvestedSource. A specialization of *DegradableEnergySource* representing energy sources that can (i) terminate at any time, and (ii) harvest additional energy in some way. The *harvestingEnergyRate* attribute describes the rate at which the energy source can harvest additional energy and add it to the amount of energy currently available to the node; the unit of measure of this attribute is Joule per second. As an example, in NODEML batteries coupled with a solar panel can be represented by an harvested energy source.

Sensor. 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 analysed and manipulated by a micro-controller. 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, etc. In NODEML a sensor has two attributes: *energyConsumptionPerSample* represents how much energy (in Millijoule) does the sensor needs to perform a single measurement, *idleEnergyConsumption* represents how much energy (in Millijoule per second) does the sensor consumes when it is not performing any reading activity.

Actuator. It is the hardware component that can physically operate on the environment. Conceptually, it performs the inverse operation of a sensor, i.e., sen-

sors acquire information from the environment and allow the micro-controller to perform some computation with it, whereas actuators are triggered by some computation of the micro-controller and then make 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, air screws, etc. Similarly to *Sensor*, the Actuator metaclass has two attributes: namely *energyConsumptionPerAction* and *idleEnergyConsumption*; their semantics is analogue to that of their sensing counterparts.
RFCommunicationDevice. A radio device to communicate with other WSN nodes, like for example the ChipCon 2420[3]. Technically an *RFCommunicationDevice* 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 [8]. Typically, those kinds of transceiver are designed for low-power and low-voltage wireless applications. In NODEML, an *RFCommunicationDevice* has the following attributes:

- *transmissionPower*: the transmission power of the transceiver in dBm;
- *receiveSensitivity*: the minimum level of a radio signal (in dBm) at which the information contained in the signal may be picked up by the transceiver;
- *frequency*: the frequency in MHz of the transceiver, it is the only mandatory attribute in *RFCommunicationDevice*;
- *antennaGain*: the degree of directivity of the antenna's radiation pattern in dBd;
- *modulation*: the name of the modulation method supported by the transceiver (e.g., PSK, APSK, ASK);
- *encryption*: the name of the encryption algorithms supported by the transceiver (e.g., AES-128, CTR, CCM).

Memory. It represents any kind of memory storage in the WSN node. In NODEML a memory element has a specific *size* in Kilobytes, an *averageReadingEnergyConsumption* that represents how much energy in MilliWatts is necessary to perform a read operation from the memory, and an *averageWritingEnergyConsumption*

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

that represents how much energy in MilliWatts is necessary to perform a write operation to the memory.
VolatileMemory. A special kind of memory representing 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 [9].
StorageMemory. A special kind of memory representing the storage device of the WSN. This kind of memory is usually utilized for persisting data within the WSN. Differently from 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 [9]. According to NODEML, a storage memory can be of different types: *configuration*, *archive*, and *program* memory, depending on how the application uses it. More specifically, configuration memory is used for storing configuration data or other essential information that must persist on the node (independently from its power cycle or firmware updates), archive memory is used to persist some data related to the running application, and program memory is used to store the binary program code of the application.
Micro-controller. It represents the micro-controller that can be mounted in the WSN node. 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 the following elements:

- one or more processors,
- zero or more ADCs,
- zero or more DACs,
- one or more timers,
- a volatile memory,
- a program memory,
- a storage memory,
- an optional radio transceiver.

Processor. Basically, it represents the common concept of processor in standard computer systems. It is the element which physically performs the computational logic of the node. Examples of processor include: 8 bit AVR Mega, ARM920T, ecc. 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.
ADC. Abbreviation for Analog-to-Digital Converter, it is a device for converting a continuous physical signal into a digital value that "discretizes" it. NODEML allows designers to specify two attributes of an ADC:

- *resolution*: the number of bits the ADC uses to represent the discrete values it can produce;
- *channels*: the number of analog sources that can be connected to the ADC at the same time.

DAC. Abbreviation for Digital-to-Analog Converter, it is a device that perform the inverse operation of an ADC: it converts digital values into continuous physical signals. Similarly to ADC, in NODEML designers can specify the *resolution* and the number of *channels* of a DAC.
Timer. A device apt to periodically trigger the clock of the WSN node. It can be implemented either as a hardware or software component and usually it 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).
PowerMode. 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 power mode identifies a set of node elements (such as memory, DAC, RF comm. device, etc.) and distinguishes between 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.

3. ENVML Metamodel

Environment is the metaclass representing the root element of each ENVML model. It represents the overall area in the 2D space in which the WSN will be

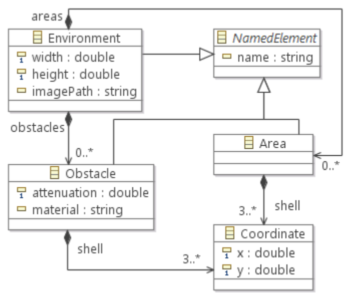

Figure 4: ENVML Metamodel

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 [10]. 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 [11]), or from other design tools. The rationale behind the *imagePath* attribute is that environment designers may provide a more detailed view 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. An environment contains a set of obstacles and areas.

Obstacle. It specifies any kind of relevant obstacle which can be placed in the physical environment. Each obstacle is characterized by the name of the *material* it is made of (e.g., concrete wall, wooden door, glass, etc.), and an *attenuation* coefficient representing of the obstacle; the latter attribute ranges 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 shape of the obstacle is given by its *shell*, it is a sequence of *coordinates* representing the perimeter of the obstacle in the 2D space.

Area. Used to identify a portion of physical environment in which nodes of the same type can be distributed accordingly to a distribution policy (defined in the DEPML modeling language). Similarly to Obstacle, the *shell* reference represents the perimeter of the area.

Coordinate. A coordinate represents a point in the 2D space within the environment. It is defined as a couple (x, y), where the point is x meters to the right and y meters up from the top-left corner of the environment bounding box; thus, in ENVML all coordinates are relative to the top-left corner of the environment bounding box, where the top-left corner has $(0, 0)$ coordinate. For each coordinate, both x and y must have positive values.

4. MAPML Metamodel

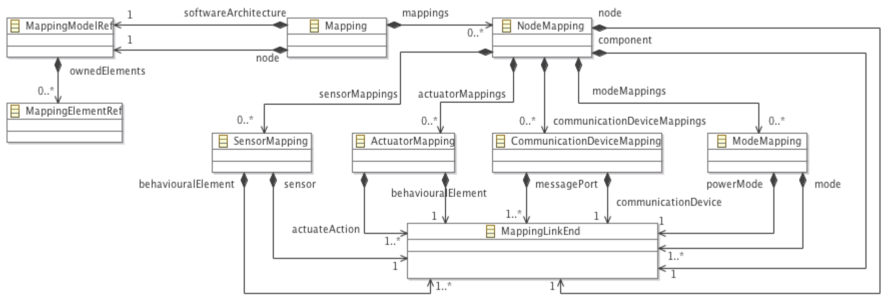

Figure 5: MAPML Metamodel

Mapping. The root of every MAPML model, it references the linked SAML model via the *softwareArchitecture* containment reference, the linked NODEML model via the *node* containment reference, and it is made of a set of node mappings (via the *mappings* containment reference).
NodeMapping. It maps a software component in SAML (see the *component* reference in the figure) to its corresponding node configuration in NODEML (see the *node* reference in the figure). A node mapping can contain a set of secondary links, each of them can be seen as a refinement of the node mapping.
SensorMapping. It 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. It is similar to the *SensorMapping* concept, but it refers to actuators, rather than sensors. So, it maps either an *Actuate* action or an *ActuatorEvent* event in an SAML component to an *Actuator* device in a NODEML node configuration.
CommunicationDeviceMapping. It 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*. Basically, this kind of link

allows designers to physically map a software port to its corresponding physical radio transceiver.
ModeMapping. It maps an energy mode defined in an SAML component to its corresponding power mode in the linked NODEML node configuration. This kind of link 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 energy modes in the pure "software world", independently from the physical power modes that the WSN node has in the "physical world".
MappingModelRef. An auxiliary metaclass whose instances act as proxies to the linked SAML and NODEML models. It is used by instances of the *Mapping* metaclass in order to refer to the linked models. Basically, we use it to achieve a strong decoupling between the MAPML metamodel and the metamodels it links; indeed, with this solution we are not polluting the MAPML metamodel with classes coming from external metamodels (i.e., Component and Node in SAML and NODEML metamodels, respectively).
MappingElementRef. An auxiliary metaclass whose instances act as proxies to elements of the linked SAML and NODEML models. This class has been introduced for the same reason why we introduced the *MappingModelRef* metaclass (see above for the details).
MappingLinkEnd. An auxiliary metaclass which is used to specify the end points of every mapping link in the MAPML model (i.e., instances of the *NodeMapping*, *SensorMapping*, *ActuatorMapping*, *CommunicationDeviceMapping* and *ModeMapping* metaclasses). We need the *MappingLinkEnd* metaclass for technical reasons; basically, we use it in order to do not directly refer to *MappingElementRef* instances in the mapping links. In so doing, it is possible to refer the same external model element by different link ends, without the need to replicate their corresponding element reference.

5. DEPML Metamodel

Deployment. The root of every DEPML model, it references the linked NODEML model via the *node* containment reference, the linked ENVML model via the *environment* containment reference, and it can contain a single type of link: *DeploymentLink* via the *links* containment reference.
DeploymentLink. It links together a node configuration in NODEML (referenced by the *nodeType* reference in the metamodel) and an area in ENVML (referenced by the *area* reference in the metamodel). The semantics of the *DeploymentLink*

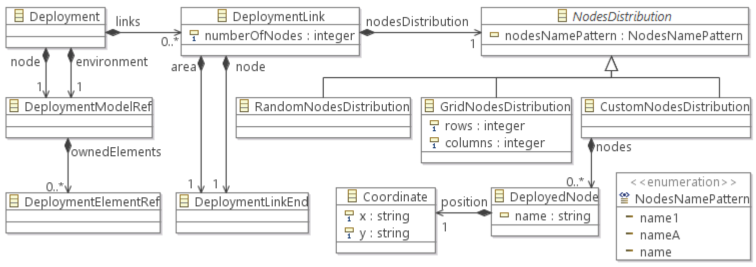

Figure 6: DEPML Metamodel

concept is that the linked node configuration is (i) instantiated and (ii) 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. Designers must also specify how the instantiated nodes are distributed within the area by setting the *nodesDitribution* reference of type *NodesDistribution*.

NodesDistribution. Abstract metaclass representing how nodes of a given type are positioned within an area. It contains an attribute called *nodesNamePattern* for declaring the textual pattern of the names of the nodes distributed within the area, they are:

- *name1*: the name of each node instance follows this pattern: $< name >< number >$, where *name* is the name of the NODEML Node element linked by the deployment link and *number* is a unique incremental integer number.

- *nameA*: the name of each node instance follows this pattern: $< name >< letter >$, where *name* is the name of the NODEML Node element linked by the deployment link and *letter* is a set of capital letters along the English alphabet.

- *name*: the name of each node instance simply corresponds to the name of the NODEML Node element linked by the deployment link; it is important to note that in this case there may be potential node name clashes in the resulting WSN network. This type of nodes name pattern is useful when considering the base station node, which is usually a single node identified just by its name.

RandomNodesDistribution. A special kind of nodes distribution in which each node is placed randomly within the area.
GridNodesDistribution. A special kind of nodes distribution in which nodes are placed on a grid with a certain number of *rows* and *columns*.
CustomNodesDistribution. A special kind of nodes distribution in which each node is manually placed within the area. Therefore, an instance of the *CustomNodesDistribution* metaclass contains *numberOfNodes* instances (see this attribute in the *DeploymentLink* metaclass) of the *DeployedNode* metaclass, each of them defining the exact position of the node instance within the WSN.
DeployedNode. It represents an instance of node within a an area with a custom distribution of nodes. The node is defined by its *name*, which must be unique within the area and the coordinates of its *position*.
Coordinate. A coordinate represents a point in the 2D space within the environment. For more details, please refer to the description of the *Coordinate* metaclass in the ENVML metamodel. The point identified by an instance of this metaclass must be inside the boundaries of the linked area in the ENVML model.
DeploymentModelRef. An auxiliary metaclass whose instances act as proxies to the linked NODEML and ENVML models. It is used by instances of the *Deployment* metaclass in order to refer to the linked models for the same reason why we introduced the *MappingModelRef* metaclass in the MAPML metamodel.
DeploymentElementRef. An auxiliary metaclass whose instances act as proxies to *elements* of the linked NODEML and ENVML models. This class has been introduced for the same reason why we introduced the *MappingModelRef* metaclass in the MAPML metamodel.
DeploymentLinkEnd. An auxiliary metaclass which is used to specify the end points of every mapping link in the DEPML model (i.e., all the indirect instances of the *DeploymentLink* metaclass). This class has been introduced for the same reason why we introduced the *MappingLinkEnd* metaclass in the MAPML metamodel.

References

[1] C. Szyperski, Component Software. Beyond Object Oriented Programming, Addison Wesley, 1998.

[2] P. Levis, D. Gay, TinyOS programming, Vol. 1, Cambridge University Press, 2009.

[3] A. Dunkels, B. Gronvall, T. Voigt, Contiki-a lightweight and flexible operating system for tiny networked sensors, in: Local Computer Networks, 2004. 29th Annual IEEE International Conference on, IEEE, 2004, pp. 455–462.

[4] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, R. Han, Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms, Mobile Networks and Applications 10 (4) (2005) 563–579.

[5] Q. Cao, T. Abdelzaher, J. Stankovic, T. He, The liteos operating system: Towards unix-like abstractions for wireless sensor networks, in: Proceedings of the 7th international conference on Information processing in sensor networks, IPSN '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 233–244.

[6] I. Demirkol, C. Ersoy, F. Alagoz, MAC protocols for wireless sensor networks: a survey, IEEE Communications Magazine 44 (4) (2006) 115–121. doi:10.1109/mcom.2006.1632658.

[7] J. N. Al-karaki, A. E. Kamal, Routing techniques in wireless sensor networks: A survey, IEEE Wireless Communications 11 (2004) 6–28.

[8] D.-M. Han, J.-H. Lim, Smart home energy management system using ieee 802.15.4 and zigbee, Consumer Electronics, IEEE Transactions on 56 (3) (2010) 1403 –1410.

[9] L. Mottola, G. P. Picco, Programming wireless sensor networks: Fundamental concepts and state of the art, ACM Comput. Surv. 43 (2011) 19:1–19:51.

[10] Definition of minimum bounding box (2012). [link].
URL `http://www.mathopenref.com/coordbounds.html`

[11] AutoCAD website (2012). [link].
URL `http://usa.autodesk.com/autocad/`