



FP7-ICT-SEC-2007-1
Contract no.: 225186
www.wsan4cip.eu



WSAN4CIP

Deliverable 1.4

Tools and methods for systematic WSAE engineering

Editor:	Steffen Peter, IHP
Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	31 st March 2010
Actual delivery date:	14 nd April 2010
Suggested readers:	Consortium, Research community
Version:	1.00
Total number of pages:	64
Keywords:	WSAN, Simulation, Engineering, Design

Abstract

The goal of this deliverable is to research an application-centric communication system engineering framework that supports system engineers in analysing and defining requirements as well as providing tool support during the selection of particular network mechanisms to be used within the WSAE under development. For this purpose we present a general design flow to help engineering WSAE applications. The semantic centre is a knowledge base it allows to describe requirements, environment, components and the whole system. The presented model-driven scheme as part of the component description is able to predict the properties of the overall system based on static properties of components in context of the given environment. For more complex dynamic properties a novel simulation approach is presented that combines network simulations with actual properties of the operating system and the software stack. Operations of the development flow are supported by various tools. First examples indicate the effectiveness of the proposed design flow and promise to improve engineering of WSAE system in future.

Disclaimer

This document contains material, which is the copyright of certain WSAN4CIP consortium parties, and may not be reproduced or copied without permission.

All WSAN4CIP consortium parties have agreed to full publication of this document.

The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the WSAN4CIP consortium as a whole, nor a certain party of the WSAN4CIP consortium warrant that the information contained in this document is capable of use, or that use of the information is free from risk, and accept no liability for loss or damage suffered by any person using this information.

Impressum

Wireless Sensor Networks for the Protection of Critical Infrastructures

WSAN4CIP

WP1 CIP Requirements and methodologies

Document title: Tools and methods for systematic WSAN engineering

Editor: Steffen Peter, IHP

Work-package leader: Evgeny Osipov, LTU

Estimation of PM spent on the Deliverable: 18

Copyright notice

© 2010 Participants in project WSAN4CIP

Executive summary

In this deliverable we tackled a critical point in developing software and systems for wireless sensor and actuator networks in the context of critical infrastructure protection. We present a general design flow to help engineering WSN applications. The design flow is a general requirement-driven design and test loop which consists of three integral operations: a) top-down mapping of requirements to a reduced design space of design alternatives, b) systematic composition and implementation, and c) testing and verification the system under development.

Each operation is supported by a central knowledge base which systematically contains all information needed for the engineering process. The structure, motivated in Section 2 is finally extended to an open system-independent description language for WSN systems. It allows to describe requirements, environment, components and the whole system.

To support the usability and acceptance of the knowledge base we presented several tools to manage the content of databases. Further an open objective scheme to describe application requirements is given. The presented application requirement description scheme breaks down fuzzy requirements given by the system developers to precise measurable system metrics. They are the formal input needed for the actual application composition process. Furthermore the presented requirement definition can be applied as general language to characterize WSN application – even outside the presented tool-chain.

The proposed tool-supported composition process additionally relies on a component meta-model that combines properties from software engineering with domain-specific behaviour prediction. The presented model-driven scheme is able to predict the properties of the overall system based on static properties of components in context of the given environment. For more complex properties especially in the area of dynamic network behaviours we presented a simulation approach that combines network simulations with actual properties of the operating system and the software stack.

The examples in Section 6 showed the general practicability of the proposed approach. The radio property example includes a static channel assessment model to predict properties of a wireless channel. It highlights the possibilities to model even complex systems building on few model descriptions. The second example is related to one of the demonstrators of the WSN4CIP project. Even though it is work in progress it already shows that the presented tool-supported design flow can benefit the development process. The real value however will be demonstrated in the other technical work packages 2 to 5 of the WSN4CIP project where we expect the proposed design flow to be applied.

List of authors

Company	Author
INOV	Renato Nunes; Paulo Pereira
IHP	Steffen Peter, Krzysztof Piotrowski, Rita Winkler
LTU	Laurynas Riliskis, Evgeny Osipov

Table of Contents

Executive summary	3
List of authors.....	4
Table of Contents	5
List of Figures.....	6
Abbreviations	7
1 Introduction	8
2 Methodology of composition-driven development for CIP	10
2.1 General Design Flow	10
2.2 Key Challenges	12
2.3 Structure of the Knowledge Base.....	13
2.3.1 Practical Data Structure	13
2.3.2 Requirement Definition	14
2.3.3 Module Definition	15
2.3.4 Reasoning	19
2.4 Related Work	20
3 Evaluation of Protocol Parameters	23
3.1 Introduction.....	23
3.2 Similitude of computer model of communication systems	23
3.3 Background on TinyOS and NS-3	24
3.3.1 On TinyOS.....	24
3.3.2 On ns-3.....	25
3.4 General idea behind our approach to design of a realistic simulationframework for CIP WSN	25
3.4.1 System design	26
3.4.2 Implementation	27
3.5 Summary	28
4 Formal WSAN Description Language	29
4.1 XML System Description Language of UbiSec&Sens	29
4.1.1 Introduction.....	29
4.1.2 Schema Overview	29
4.1.3 Hardware Properties	30
4.1.4 Security Aspects	31
4.1.5 Modules Functional Grouping	32
4.1.6 Software Description	33
4.1.7 Nodes description	36
4.2 Extension of the XML language to support description of communication modules.....	38
4.2.1 Parameter Type.....	38
4.2.2 Component Description	40
4.2.3 Requirement Description	43
4.3 Examples.....	44
5 Tool support for the WSAN engineering	46
5.1 Set up of databases.....	46
5.2 Selection of Application Requirements	48
5.3 Selection and Evaluation of a Suitable Module Composition.....	49
5.3.1 Differences to the Selection in UbiSec&Sens	49
5.3.2 Selection Process	50
6 Examples	54
6.1 Radio properties	54
6.2 WSAN4CIP FWA demonstrator.....	56
7 Conclusion.....	59
References	60
Annex A Scheme of WSAN4CIP Components and Requirements	62

List of Figures

Figure 1: General Design Flow	10
Figure 2: V-Model of system development	11
Figure 3: Data Structures (data bases and documents) and processes (rounded boxes) of our WSA engineering process	13
Figure 4: Detailed MAC layer architecture as shown in [28](a)) and the corresponding representation in our composition model with components, tied interfaces, and external parameters (b)).	16
Figure 5: Visual representation of a module database: ellipses show interfaces and boxes represent components	18
Figure 6: OMG's Model Driven Architecture	21
Figure 7: In bold is shown the system part where the tos-ns implemented.	26
Figure 8: Dependences between TinyOS, ns3 and the role of TOS-NS.	27
Figure 9: TOS-NS implementation and compilation component flow	28
Figure 10: Description of a WSA	30
Figure 11: Security characterisation	32
Figure 12: Security parameters	32
Figure 13: Functional grouping	33
Figure 14: Software interfaces	34
Figure 15: Software components	35
Figure 16: Software components interfaces	35
Figure 17: Description of a node	36
Figure 18: Node hardware	36
Figure 19: Node software	37
Figure 20: Interconnection of software components	38
Figure 21: Definition of a parameter type	39
Figure 22: Parameter structure: parameters, requirements and properties inherit from the ParamType and only add few individual types.	40
Figure 23: Definition of a component type	41
Figure 24: Definition of a software component	42
Figure 25: Definition of an interface	43
Figure 26: Definition of requirements	43
Figure 27: Chain of tool-supported WSA engineering. Yellow boxes name the tools to generate the blue documents.	46
Figure 28: GUI of the components database editor	47
Figure 29: Process and data structures of the requirement definition: Application designer chooses basic requirements in a form, automatic mapping creates tables as input to further processing.	48
Figure 30: User interface for the selection of the application requirements	49
Figure 31: Example for re-convergences in the configuration graph. A combination of modules c, e and f is the smallest, considering app needs interface 1 and 2. The numbers in the brackets represent size.	52
Figure 32: Model for the assessment of radio properties: sensor node with three components communicates of a channel defined by environmental parameters. Arrows show the information flow during the assessment process.	54
Figure 33: Software architecture of the FWA sensor node	57
Figure 34: Basic model of components and interfaces intended for the FWA system	57

Abbreviations

CIP	Critical Infrastructure Protection
FEC	Forward Error Correction
GUI	Graphical User Interface
MAC	Medium Access Control
MIC	Message Integrity Code (Message Authentication Code)
OOP	Object Oriented Programming
QoS	Quality of Service
SAP	Service Access Point
SNR	Signal-Noise-Ratio
WSAN	Wireless Sensor and Actuator Network
XML	Extensible Markup Language

1 Introduction

Engineering dependable systems in the area of wireless sensor networks (WSN) is a complex task. For different application basic mechanisms are often similar but the concrete requirements of specific application enforce the application of different means or combination of means. The goal of this deliverable as conclusion of work package 1 is to research and provide an application-centric communication system engineering framework that supports system engineers in analysing and defining requirements as well as providing semi-automatic or even fully automatic support during the selection of particular network mechanisms to be used within the WSAN under development.

Literature is rich on engineering approaches for standard software systems. Unfortunately most of them do not meet the needs of WSAN development due to the different focus in the development process. Important differences to standard software development are:

Importance of environmental parameters

When systems have to be engineered for protection and surveillance of critical infrastructure, the systems and the components of the systems have to work where the critical infrastructure is deployed – and that is usually not inside a PC or a lab. Tests in the outside are expensive and time-consuming, and simulations do not always deliver satisfying results.

Novel technical requirements for most applications

New operation environments inherently mean new requirements that stem from the properties at the operation side. At least for today's implementations of CIP by WSANs there is no existing stock of working solutions nor general frameworks that can be reused. While standard software engineering is mostly about combining of well-known patterns, WSAN engineering is to a large extent about developing new basic means.

Lack of a common design language

Many description languages exist to model data streams and entity relations, while means for description of systems in a specific environment are sparse. An accepted description methodology is mandatory for the establishment of knowledge bases and good engineering.

Diversity of properties

Required properties of standard software are usually related to data pattern. In WSAN system the requirements range from physical properties (energy, radio) to behavioral aspects (timing) and are highly domain specific. Data management is just one small aspect of many.

Need for interdisciplinary knowledge

Developers of WSAN systems have to make many decisions far outside the usual domain of software engineering. In the context of CIP for instance it includes the choice of the radio system, encoding algorithms and mechanisms to provide security and robustness. Even if no new algorithms have to be developed, simply the selection of the proper means for the given scenario needs a lot of in-depth knowledge in the specific domain.

The differences to standard software as described above are especially true for the communication stack of the wireless CIP systems. Properties of the wireless channel are usually only valid for the specific operation side. One-fits-all solutions, have been proposed, but do not show good properties for most scenarios.

From our experience developers in this field do not have any structured support in the development process. They usually posit the system based on individual knowledge and experiences in a sort of artistic process. Validation of the correctness of the result is given practical tests: either it passes the tests or the program has to be refined. Such a reasoning process does not offer any assembly guide, i.e. how to choose the right components so that the composition meets the given requirements.

A classic approach to handle such complex systems are layered architectures. The hardware access, different network and routing layers at the bottom - middleware, general services and data processing in the middle - and an application layer on the top. Such architectures are very common in standard software and provide good flexibility at both development and run time. However, in the context of WSNs such layered architectures do not seem to be the best solution. Strictly separated layers need additional memory, and inter-layer communication needs additional computational efforts - two resources that severely constrained WSN devices lack.

A promising approach that can solve the issue and that has been widely accepted is to use small utterly specialized modules and connect them directly without complicated internal structure or inter-module communication. A compiler literally melts all required components to one block what promises to be very memory efficient and results in reduced computational overhead. Beside the problem of managing the large set of small modules the major issue of such programs is development and maintenance costs. The substitution of one module may affect many other modules. It is particularly the case with respect to communication and security properties. An alternation of the requirements that eventually lead to the need for changing few components would correspond to a complete new development of the system.

This finally leads to the initial problem: the missing tool support in the area of WSAN engineering. This deliverable tackles several aspects that are mandatory to establish a tool-supported engineering methodology for WSAN systems. Section 2 analyses existing work and describes a general composition-driven design flow for WSNs and CIP. Section 3 focuses on the assessment of dynamic properties by simulations. In Section 4 we propose an open system-independent description language for WSAN systems. Based on this language Section 5 presents a tangible design flow supported by several tools to engineer WSAN applications. Examples for the design flow are given in Section 6, before the document concludes in Section 7.

2 Methodology of composition-driven development for CIP

As introduced, the development of systems and software in the area of sensor networks and critical infrastructure has similarities to developments in other application areas. Basically, several software and hardware modules are supposed to work together to fulfil a specific task. However specific requirements, which were explained in the previous section, forbid the direct application of means established in the world of standard software development. This section analyses existing approaches and presents a general methodology to cope with the requirements. The section concludes with a list of general information needed for the proposed design process.

2.1 General Design Flow

This subsection introduces the general design flow we will use for the rest of this document. It is universally applicable for both manual and automatic design processes.

The fundamental idea is that application requirements will be mapped to a system, which will be implemented and finally tested against the requirements. These three steps are represented by the three ellipses in Figure 1.

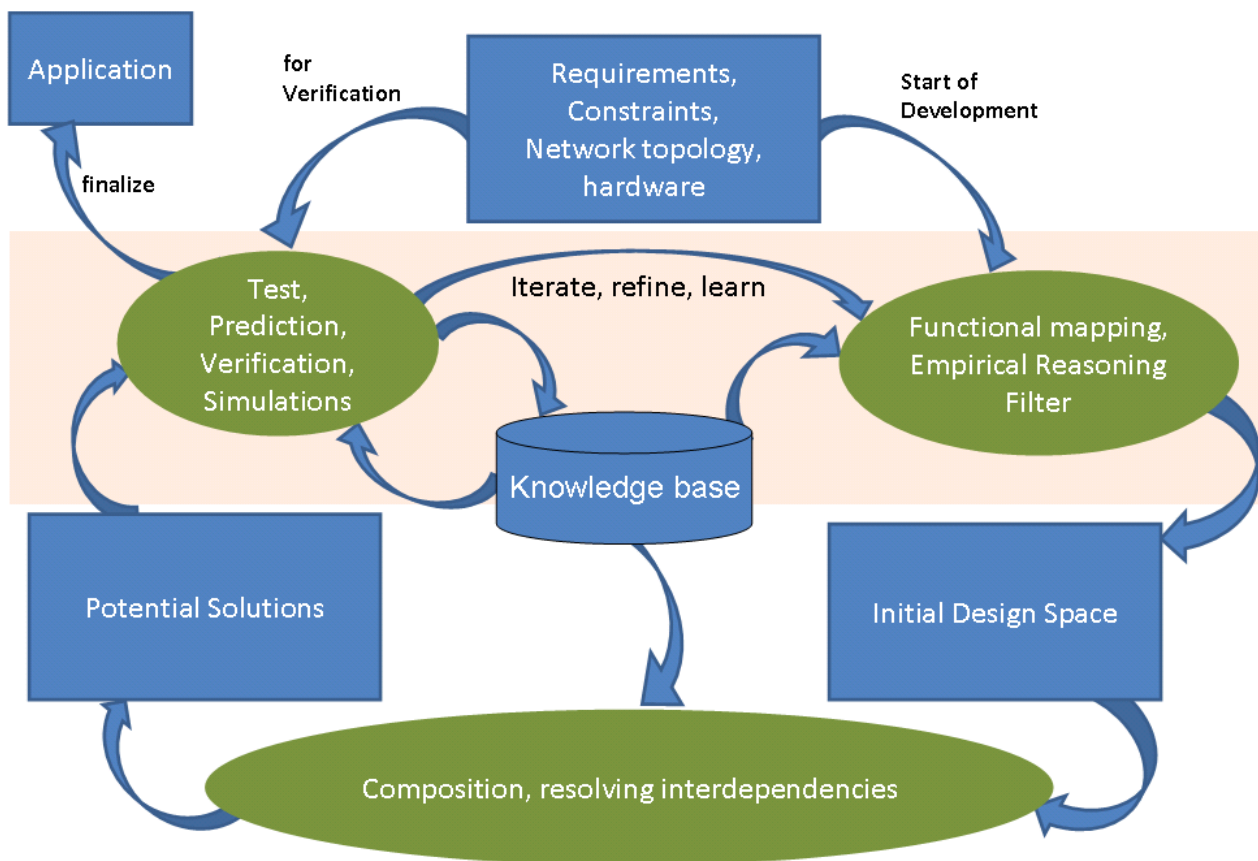


Figure 1: General Design Flow

- Starting with application requirements and using a pool of existing knowledge and techniques, a developer will map the existing knowledge and partial solutions to a first draft of the required system, in the step *Functional mapping and Empirical Reasoning*.
- In the second step *Composing and resolving interdependencies* the initial design space will be refined, completed and extended by new implementations. Beside the developer's creativity again the set of existing knowledge and solutions will be applied in order to advance.
- When finally a potential solution is found, it will be verified and tested against the initial requirements, in the step *Test, Prediction, Verification, and Simulations*. Means for those tests are lab experiments, simulations, formal verifications and tests in practise. If the system passes the tests, it can be considered as solution for the initial problem. If it fails, the development has to go back to step two or even step one. Reiteration of the development steps under consideration of the test results are supposed to eventually conclude in a system that passes the tests.

In particular in the field of software engineering countless models have been presented to refine that general flow. For example the similarities to the V-Model (Figure 2) as it is known in engineering of embedded systems and software in general are apparent. There starting with the specification the top-down design process will conclude in an implementation. Implementation will be integrated and tested (bottom-up) until the development process concludes in the final application that matches the initial specifications. Verification and feedback to the design phase during integration and test ensures the correctness of the implementation process.

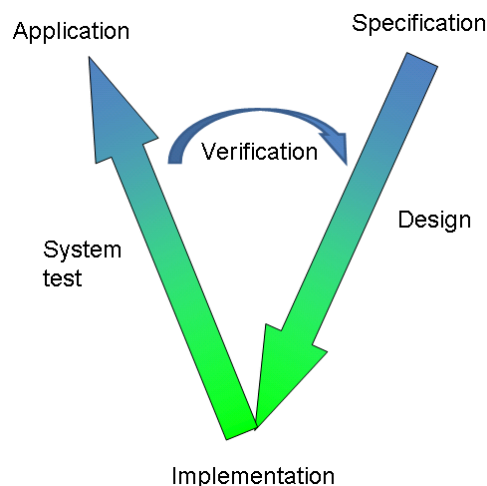


Figure 2: V-Model of system development

The model in Figure 1 contains additional details that already aim towards the tool-support presented later in this document. On the right side we have the top-down (traditional) reasoning that maps requirements on modules based on empirical data and experiences. It does not necessarily lead to a full final solution but to a smaller design space. The smaller design space will be starting point for the composition process that looks for potential solutions. Logically it corresponds to the implementation phase of the V-Model, while it allows automatic composition.

Finally, the set of potential solutions has to be tested and evaluated against the initial requirements. Here we rely on functions and heuristics stored in the knowledge base. So knowledge and properties will be accumulated bottom-up from the components to the application. Tests and verification can be either static or dynamic. Static tests can be used for simplified heuristics and reasoning. Means for dynamic testing are simulations or real world tests. Tests and verification play an important role in the design flow for two reasons: first, obviously they validate a proposed configuration. Second, even if a configuration fails, the tests results provide valuable information for the system under development but also for future systems.

2.2 Key Challenges

With the general design flow we can state WHAT we want to do in each step. Even though inputs and output of the steps are defined it is not obvious HOW it can be done. In this subsection we identify the key challenges within the proposed general design flow.

Technical description of the requirements

Requirements are needed as initial driver of the design phase and for the final testing. To have an objective and reproducible engineering process, the requirements have to be objective and reproducible.

Mapping of the requirements to verifiable measurable metrics

An objective technical description of requirements does not automatically correspond to measurable and computable set of metrics as it is required for the verification process. For instance a topology description is not measurable. Also the number of nodes or the distance between them is neither measurable nor predictable as software metric.

It is necessary to combine and map the requirements from the first item to a list of measurable properties and given assumed environmental conditions. Such measurable units would be data rate, delivery ratio, latency, energy consumption.

Early prediction of the behaviour of assembled modules

Since it is the goal to test the properties of the system before deploying the eventual system, we need tools to predict the eventual behaviour based on modelling. Certain properties can be estimated on rather high level. Examples are the theoretical data rate and latency for the given topology under the assumption of a simplified wireless channel. More complex simulations of actual channel properties need complex simulations in specialized software

Composition process

Basically the problem statement of module composition is trivial: after trying all possible combinations the best one is selected. Beside the question how to assess the 'best one', which is already addressed in the early-prediction-of-behaviour-challenge, the main challenge here concerns the runtime of the selection process. With a large number of potential components, evaluation of all possible combinations is not feasible, so optimisations will be imperative.

Formal meta-description of components

Reproducible module composition needs a formal framework it can work with. Without such a frame it would not even be possible to compare two components reasonably. Meta-description approaches are well-accepted means to describe syntax and semantics of components. The challenge is to find a suitable description framework that provides the functionality we need to describe the properties we need, while not increasing the complexity of the framework to an unbearable level.

Objective and verifiable testing process

Regardless of automatic or manual engineering, testing and verification of potential solutions has to be reproducible and significant. It should be possible to derive tests cases from the requirements, and the test cases have to return specific measurable results.

The issues on how the design flow can be realized can be grouped into two sub-problems: First, the question how can we store and represent the knowledge that is part of the knowledge base, and second, how to compute the implications based on this knowledge base. In fact the organisation of the knowledge base appears to be the most critical part in the proposed engineering methodology, because in the moment one knows why to choose a specific configuration (it is a data issue) it is straightforward to define an algorithm to find the configuration.

2.3 Structure of the Knowledge Base

As pointed out in the previous section, the structure of the knowledge base is the most critical point for the systematic WSAE engineering. Once we can document why specific design decision have been made, the decision are practically reproducible. For this purpose this subsection investigates the presented design flow from data perspective. Required information bases will be identified and a first draft of needed data items will be elaborated.

2.3.1 Practical Data Structure

Figure 3 shows the information flow between the main functions of the selection process. The data exchanged between the function steps is illustrated as file, while the constant back-end of the selection process is illustrated as data bases. In this section we introduce the data structures needed for the decision and selection process. Even though the described structures already aim towards the tool support we present later in this document, at this point the information are intended to be applicable also for a classic manual design process.

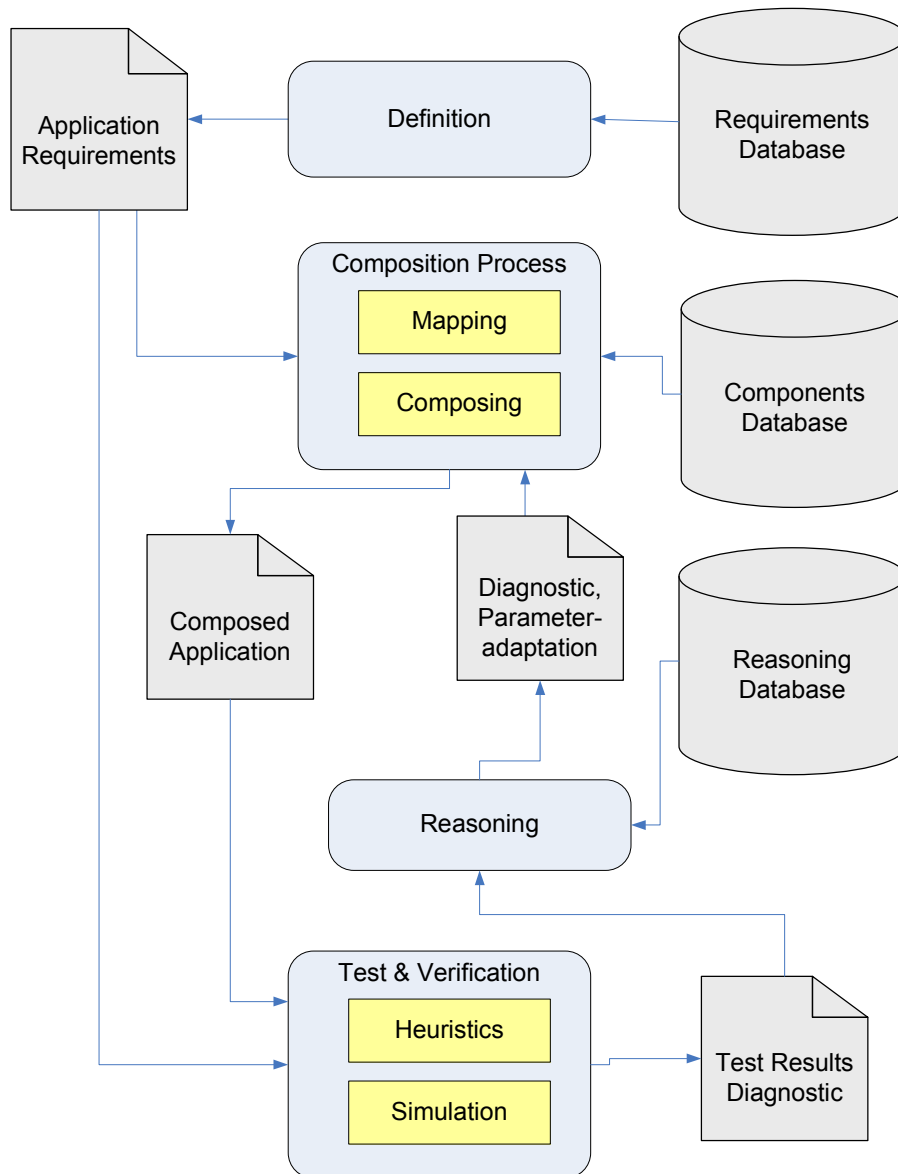


Figure 3: Data Structures (data bases and documents) and processes (rounded boxes) of our WSAE engineering process

In the following subsections the three document types will be introduced classified regarding their general type: requirements, components, and reasoning.

2.3.2 Requirement Definition

The definition of requirements is one key issue during the development of any kind of system. Only with the precise information of what should be achieved it is possible to perform precise and goal-oriented engineering. To the best of our knowledge currently no accepted and reliable way of defining WSN application and their parameters is known. A major problem is the diversity of application domains and the resulting huge space of variables. For example [6] introduced an approach to characterise WSNs by using 12 orthogonal major dimensions, some using several sub dimensions. Another problem is that experts in the domain are usually not familiar with the terms used in WSN engineering. So what is needed is a translation of rather high-level, fuzzy, domain-specific requirements to measurable metrics that can be tested within the WSN development process.

As illustrated on top of Figure 3 our approach uses a central Requirement database. It contains all requirements a designer can possibly chose. It is a sort of a table that characterises the requirements, their domain and design-space and the connection to other requirements. In contrast to [6] domains are not used as dimensions but primarily as an attribute to filter properties. For example all network properties are grouped by the domain attribute 'network'.

In the Definition phase an application designer will chose a set of requirements and set the parameters. The document containing the parameterised subset is a first version of the application requirements document. Due to the formality of the database this document is already a well-defined application description. However, the value for the selection process is still limited because environmental parameters and requirements of performance parameters are just mixed in one large table.

We believe that it is necessary to separate requirements in global parameters, which are straight constraints of the system, and in measurable performance metrics. The latter have to be measured, simulated or derived as result of the system development process, in order to check the suitability.

With other words: if G are global constraints, and C is the development under development then $f_p(C, G)$ defines a function to determine the value of the performance metric P . Thus the performance metrics have to be compared at the end of the development process, while the set G is an invariant during the process.

The separation of the two groups out of the set of designer-defined requirements will be an automatic process that should be done based on data entries inside the application database. We define the process in Section 5.2.

An additional function of the application definition process in our proposed scheme is the self-expansion of the requirement space. It is an answer to the issue of ambiguous requirement definitions. For example if the designer expresses one requirement as period (for example in seconds), solutions working with a frequency parameter (1/s) are not directly applicable. The requirement space expansion phase will expand the given requirements to all requirements that can be expressed with the given parameters. This is done by a relation-formula as part of the requirement entries in the database. By this we are convinced also to solve the problem caused by domain specific definitions. By the self-expansion the requirements will be automatically extended to the domain space of WSN engineers, considered indeed the translation has been defined before.

The data structures needed for the requirement definition and database are defined in the following:

2.3.2.1 Application Requirements

The application requirements document is the specification of the system under development. It contains the performance properties that have to be met, the constraints for the design process (environmental, economical, technical). It has also to contain the relations between the requirements and the test cases that eventually must pass. It is the target that the Application Requirements are sufficient to develop the system and to verify its correctness without additional user input.

Basically the Application Requirements document should be a list of single requirements. Each requirement should have the following data items:

- Identification (name, domain, type, id)

- Unit
- Required/defined value
 - o Either a fixed value
 - o Or a range (MIN, MAX)
- Formula or description that shows interrelationships to other properties
- Information if the value is an environmental parameter or a measurable parameter that will be determined by the composed system

2.3.2.2 Requirements Database

The Requirements Database contains all possible requirements and constraints a designer can set for an application. With other words: if a property is not in the Requirements Database, that property cannot be set for the application. It may appear a bit formalistic, but the idea is that if a property is not part of a formal reusable database it can hardly be supported in the later composition, testing and reasoning steps.

Since Application Requirements are a parameterised sub set of the Requirements Database, consequently the Requirements Database basically has to contain all properties needed in the Application Requirements. The major difference is that values are not fixed, but define a space for the properties.

Each item of the list of possible requirements should contain the following information:

- Identification (name, domain, type, id)
- Unit
- Possible design space of values
 - o Choice as selection of a given set
 - o Range (MIN, MAX, Resolution)
- Formula or description that shows interrelationships to other properties
- Information if the value is an environmental parameter or a measurable parameter that will be determined by the composed system

2.3.3 Module Definition

A common concept to cope with the complexity of the design process is abstraction. We use rather high level abstraction for both, components and interfaces.

Figure 4 shows the representation of a MAC layer architecture as shown in [28] on the left and our corresponding representation on the right. The left structure apparently is much closer to an actual implementation. Interfaces and entails of the component are shown and modelled in detail.

In our model (Figure 4b) the interfaces, which consist of several functions and ports, are bundled to few general interfaces. For example the MAC module provides just one characteristic Mac interface (MacI) instead of explicit ports for sending, receiving and additional control. With this one interface represents that group of ports and functions. The Parameters for the configuration of the module are not part a dedicated interface but will be set by at design time from extern.

Internal behavior of the components will be modeled by abstracted parameterized properties too. It is the goal to describe the functions and behavior of the modules based on rather high level heuristics, where applicable. We presume that most properties of a module can be expressed as small piece of function or code as part of the model. Indeed, it is a trade-off between the extent of the description and the quality of the results. A short description in a static model will not provide the same degree of details one will get for a dynamic simulation based on the actual source code.

Anyway, the basic concept described in this section is valid for both extremes. The interfaces shown in Figure 4b can be split up to the level of detail as seen in Figure 4a, and the internal component model can be made similar, too.

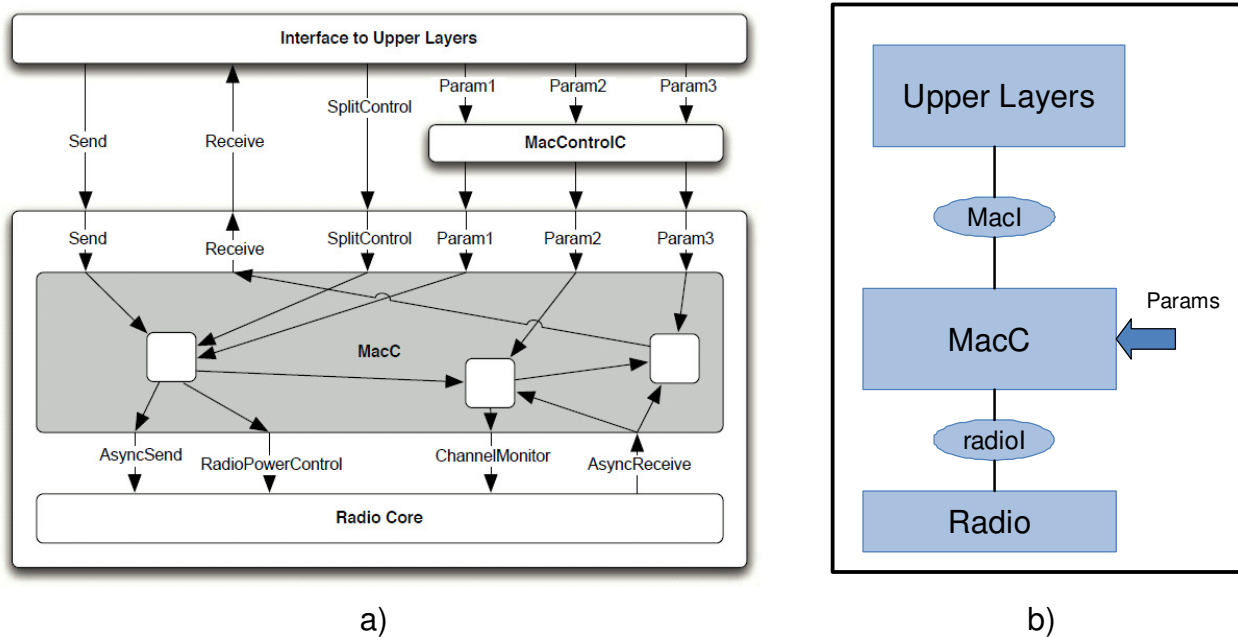


Figure 4: Detailed MAC layer architecture as shown in [28](a) and the corresponding representation in our composition model with components, tied interfaces, and external parameters (b).

The three descriptions are the input for the selection process that computes a selection of modules that work together on the specified hardware and satisfy the application requirements.

2.3.3.1 Components Database

The Components Database has to contain all possible design options and the meta-information needed to assess the implications of the composition. Practically the database contains

- a list of components with their properties
- a list of interfaces as connecting sockets between the components

Technically interfaces can be considered as ports of the components. Then it is not necessary to list them explicitly in a dedicated external list. One could connect one port of one module to the corresponding port of another module.

In our model however, interfaces are not just ports. Semantically they are service access points (SAP) of the components and they implicitly represent a service description. For example a software module with the functionality of encryption would provide the service (i.e. the interface) Encryption. Each other module that needs (uses) that service could connect to the interface without knowledge of the actual component that provides the functionality. By that interfaces are an abstraction from the actual technical implementation.

Abstraction also allows to use the inheritance concept as it is known from Object Oriented Programming (OOP) for the interfaces. For example a block cipher like the default AES in codebook mode provides a block cipher encryption. Logically it is indeed an encryption. We express it by giving the interface 'block cipher' the attribute saying that it is a special instance of 'Encryption'. That way each module that needs 'Encryption' can use the special 'Block Cipher'.

Interfaces contain the following information

- Identification (name, domain, type, id)
- Description
- Is (Inherit from another interface)
- Possible parameters
 - o Choice as selection of a given set
 - o Range (MIN, MAX, Resolution)

Parameters can be used to parameterise the interfaces. In case of encryption it could be required that a module needs encryption with a specific key length. Then the key length would be a parameter of the interface and only components could be connected that provide the interface with the required characterisation.

The basic structure of the actual components is similar to the interfaces. They need

- Identification (name, domain, type, id)
- Description
- Is (Inherit from another module)
- List of possible parameters
 - o Choice as selection of a given set
 - o Range (MIN, MAX, Resolution)

Additionally components need to list

- Interfaces they Provide
 - o parameters and constraints for the interface
 - e.g. the max number of components that may use the interface
- Interfaces they Use
 - o Indication whether the use of the interface is mandatory or optional
 - o parameters and constraints for the interface
- Properties they provide
 - o Formula, heuristics or value that sets the specific property based on properties of attached modules, environmental parameters, and parameters set for the component.

The difference between parameters and properties is that parameters will be set from the outside. During the design process the developer will make decisions to set the parameters of the components. The properties will show the ramifications of those decisions. For example a designer can chose the modulation of a radio module. That is one parameter. Data rate, energy consumption, sensitivity are values which base on that parameter. Hence they are properties of the module which can be expressed by small formulas, or the data sheet of the radio module.

An example of network-related components, their interface and the relations are shown as Figure 5. Rectangles are the components. The semantic of the arrows is as follows:

- Interface \rightarrow Component: the component provides the interface.
- Component \rightarrow Interface: the component uses the interface. If dotted, the usage is optional
- Component A \rightarrow Component B: A inherits from B (IS relation of components)
- Interface A \rightarrow Interface B: A inherits from B (IS relation of interfaces)

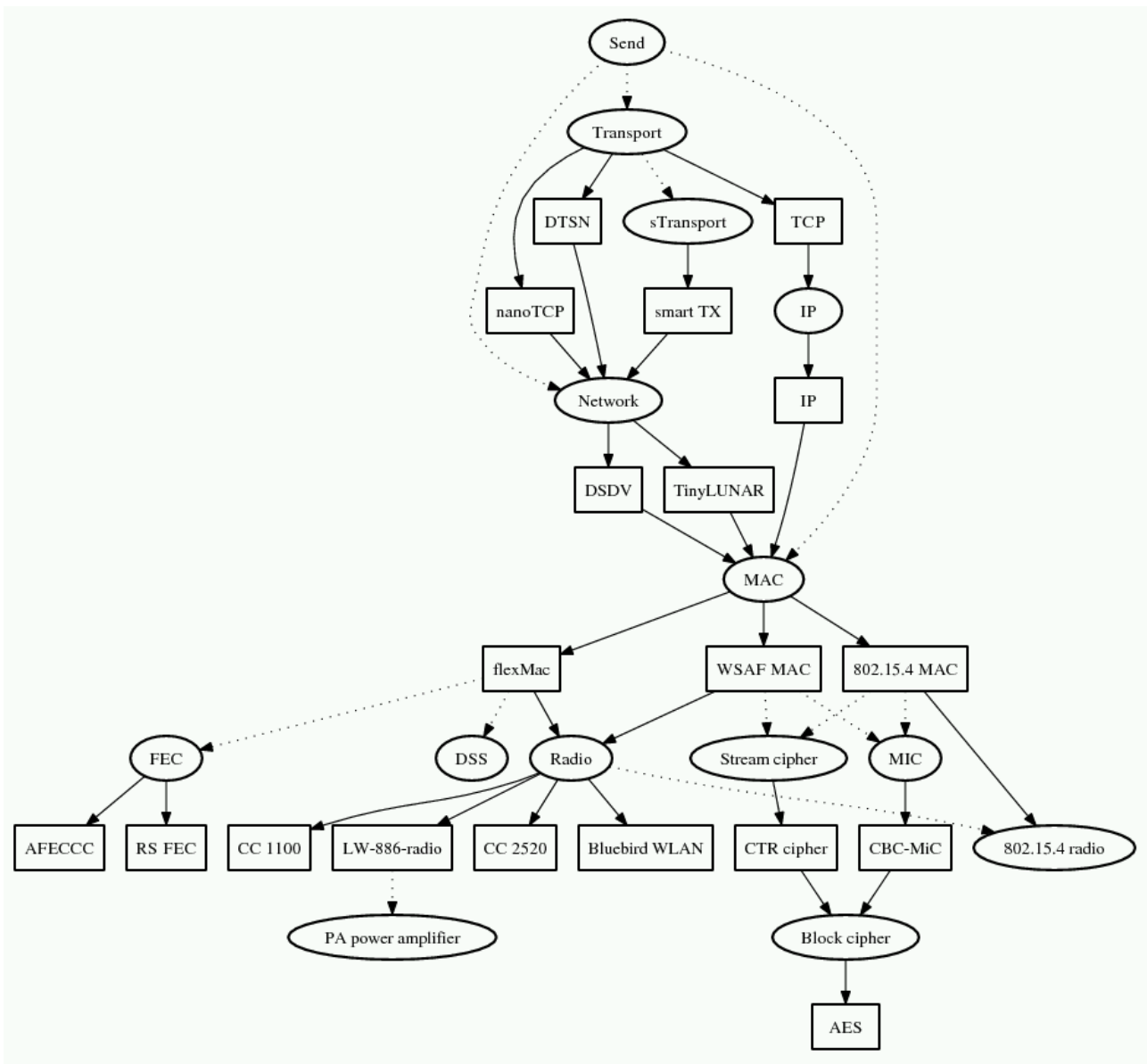


Figure 5: Visual representation of a module database: ellipses show interfaces and boxes represent components

The visualisation on this scope is similar to the other uses/provides approaches, for example as presented in UbiSec&Sens. The internal semantic however has changed significantly as we will see in 5.3.1.

2.3.3.2 Composed Application

The data structure of the Composed Application should contain all information required to describe the system. It is a sub set of the components and interfaces plus information how the components and interfaces are connected and how they are parameterised.

The actual source code of the system would be an instance of the composed application. We however focus on rather high level representations of the composition.

Interfaces inside the composition have to provide the following information:

- Identification (name, domain, type, id inside components database)
- Id for identification inside the composition
- set parameters as fixed values
- Id's of components that use the interface

- Id of the component that provides the interface

Components as part of the composition have to provide the following information:

- Identification (name, domain, type, id inside components database)
- Id for identification inside the composition
- set parameters as fixed values
- properties, still as formula – they have to be resolved as part of the test
- used interfaces
 - o id inside composition
 - o set parameters for the interface
- provided interfaces
 - o id inside composition

2.3.4 Reasoning

2.3.4.1 Test Results

The test results document is basically one protocol containing:

- Name, type, domain, unit of the property that was tested
- Required result
 - o Fixed value
 - o Or range (MIN, MAX)
- Computed result
 - o Fixed value
- Evaluation result
 - o Requirement met (true/false)
 - o Quantitative result (requirement met by fixed value)
 - o Qualitative assessment

The evaluation results can partly be deduced from the required and computed results. The qualitative assessment however is not a simple subtraction but needs a translation of the results. For example if the required data rate is 50kb/s and the computed data rate is 55kb/s, the assessment would be +1 on scale from -5 to +5, meaning that the requirement has been met – but very close. Developer and future refinement steps would recognize that there is no room to jeopardise that property. The formula for the translation from quantitative results to qualitative assessment most likely depends on the specific requirements. In future could be part of the requirements database or will be part of the reasoning database.

2.3.4.2 Reasoning Database

The reasoning database provides some help in mapping the test results to proper adaptations of the settings for the next iteration of the composition process.

For human developers that reasoning database is experience. But experience often is biased and not objective and not reproducible. The reasoning database is simply a table that lists

- what went wrong (symptom),
- possible reasons (diagnosis)
- approaches to resolve the problem (treatment)

Diagnostic and Parameter adaptation are a direct application of the approaches to resolve the problem. They are expressed as small formula describing either an adaptation of specific parameters.

Possible data entries it could be:

For the symptom 'data rate too low' we can find the possible reason 'pk_size too small' with the treatment 'increase packet size'. An alternative reason is 'sleep cycle too long' and the treatment would be 'decrease sleep cycle period'.

Currently no similar approach comparable to this reasoning database is known. Thus it is a novel concept we want to test in the integration work of the WSAN4CIP demonstrator applications. We expect to adapt and extend the structure of the reasoning database during this process, while we are convinced that already the existence of such a structure will improve the quality of engineering and the reproducibility of design decision substantially.

2.4 Related Work

Composition of components based on required-provided relations is a well-known approach in software engineering. That architectural composition approaches focus on the question whether specific properties hold in the composed system. That even includes properties related to behavioural prediction. Their context however is mostly to internal behaviour of the software [1] [2] [3]. The approach presented in [3] is positioned on a higher layer but similar to our targets. Based on a requires-provides scheme they compose components to run time. While that does not correspond to our intended target area, the verification process based on the included policy rules is applicable. For example access and flow control can be specified in independent components. A rule execution engine determines the eventual policies of the composed system and either accepts or rejects the proposed composition. The description language is CLP

[17] proposes another model-integrated development concept for embedded systems. They use domain-specific models that represent the software and the environment it operates in. The models allow the formal analysis and the verification of the software system at design time. Synthesis of source code is supported by model-based generators. They allow to generate and maintain actual implementations. It is a general framework that can be extended by domain-specific languages and tools.

The Model Driven Architecture (MDA) [9][10] (Figure 6), as defined by the Object Management Group (OMG) [11], is "a new way of writing specifications, based on a platform-independent model. A complete MDA specification consists of a definitive platform-independent base Unified Modeling Language (UML) model, plus one or more platform-specific models and interface definition sets, each describing how the base model is implemented on a different middleware platform".

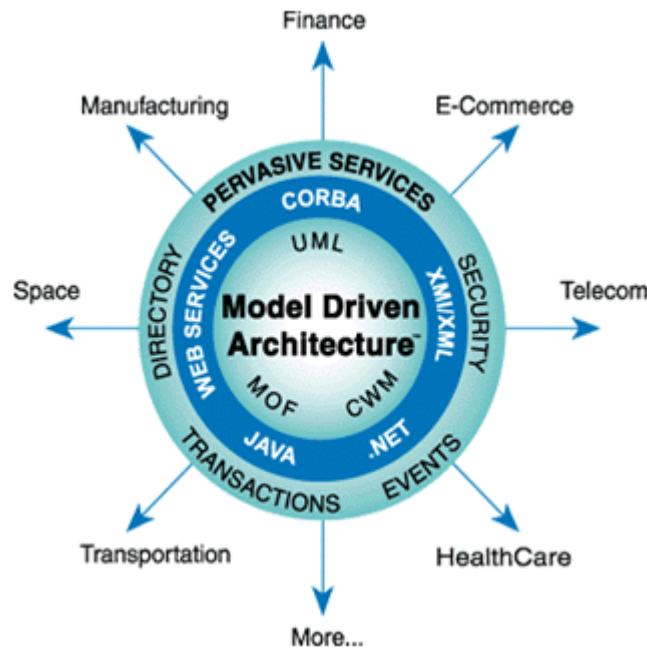


Figure 6: OMG's Model Driven Architecture

The MDA separates business and application logic that contains less complexity from underlying implementation technologies by presenting the static and dynamic aspects of a system as a high level abstraction with the implementation details hidden. The abstraction is called platform-independent model (PIM). The MDA also supports platform-specific model (PSM), which contains enough implementation information that can convert it to particular source codes. Using the concept of models, the MDA can potentially reduce the life cycle of software development and allow for reusing system modules since it is not necessary to design a completely new system with similar functionalities when a new implementation technology becomes available. A platform-independent model representing the conceptual behaviour of a system can be reused in various software environments because it does not restrict itself to any specific technology at a design stage of a software development.

Meanwhile, raising levels of abstraction from implementation details increases the readability of a system and instead of analysing complex source codes, users can view a system design made up of models, instead of specific programming languages, at a high perspective. Moreover, it saves considerable time to modify systems at a conceptual level before actually putting a design into implementation. Intuitively, models are the pillars to support the MDA and they represent the conceptualisation of real world, while source codes denote the actual implementation of a design. Then the questions raise here: how to transform an abstract visual design to detailed textual programming languages and what are the supporting technologies to develop transformations between models without losing any consistency in a life cycle of system development?

Researchers have recognized model transformation that bridges the gaps between models at different abstraction levels or between models and source codes as the heart and soul of the Model Driven Architecture. Intensive research has been conducted on the process of model transformation. Various transformation languages and tool suites have been developed, although most of them are at experimental stage yet to be applied to industrial practice. An example of such tools is the Eclipse Model Development Tools [12] and the Eclipse Modeling Framework Project [13]. Naturally, the model transformation process is complex, requires good expertise as the models are complex and some are still under development, and there are no criteria for ensuring the consistency and accuracy of the transformation between the different abstraction levels.

The MDA was conceived for object oriented design, supporting class specification, class relations, class manipulation and interchange. So, it is difficult to apply the MDA to Wireless Sensor and Actuator Network (WSAN) design. However, a few relevant works can be found in the literature:

[14] presents $XOBE_{\text{SensorNetwork}}$ (XML Objects for Sensor Networks). This work shows how to use the Extensible Markup Language (XML) [8][14] to store, process and query data on WSAN. The XML structures can be used to generate code to store the data, integrated into the C programming language. Additionally, code to process queries and aggregate query results is also produced. Naturally, to meet the hardware restrictions of

sensor nodes, XML compression is used to improve efficiency and save energy, by reducing the information transmitted in messages. XML Template Objects are used for this purpose.

[15] presents the Global Sensor Networks (GSN) platform which provides a scalable infrastructure for integrating heterogeneous sensor network technologies using a small set of powerful abstractions. GSN supports the integration and discovery of sensor networks and sensor data, provides distributed querying, filtering, and combination of sensor data, and offers dynamic adaptation of the system configuration during operation through a declarative XML-based language.

[16] presents Scatterclipse, a model-driven Eclipse based tool chain for developing, testing and prototyping Wireless Sensor Networks. Since the tool-chain is based on Eclipse, it offers a plug-in oriented architecture and is comprised of free open source components. The tool integrates visual automated application debugging and WSAN-management features in the model-driven software development process of the deployed sensor boards. The high degree of automation accelerates the development and testing of applications, which are already running on sensor nodes. Furthermore, substitutability and reusability of the software artifacts are increased, because the artifacts, alongside the automated code generation, are represented by their respective models. Both increase the development process's productivity. The model driven code generation is used to furthermore generate a largely tailor made code, so that only the required amount of code is generated for the sensor node's intended roll. Thus the scarce memory space is not only optimized, but also unnecessary calculating and energy intensive software modules are avoided. The decreased portion of manually written code also reduces the possibility of a programmer's careless mistakes. Although powerful, the tool only produces code for the sensor boards developed by the author's research group.

The use of model development tools for WSAN is still in its infancy, with a few, limited, prototypes already existing. It is expected that, as standards in this area are completed and matured, new, more powerful tools, will appear.

3 Evaluation of Protocol Parameters

3.1 Introduction

The material presented in this section stems from the component based design approach to development of communication protocols for wireless sensor networks presented in Deliverable D1.3[19]. Application of the component based paradigm to the design of network protocols was not in question until the beginning of nineties. By this time the layered model was a standard development approach. Certain design limits of the layered architecture were recognized in number of articles, e.g. [27][32]. The component based approach to the development (CBD) of software in wireless sensor networks was adopted more or less from the time of their appearance on the technology arena [26]. In addition to the simplicity of application development due to straightforward abstractions for programmers, one of the major motivating factors for adopting CBD was the *efficiency of the resulting code base*. Efficient code base is achieved by exclusion of unnecessary, for a particular application, components. A straightforward extension of the paradigm to the design of communication protocols [28] aimed at *reducing hardware dependency* of particular protocols.

Further, an ability to consciously select appropriate software components for combining in an application-specific protocols stack is essential in the domain of wireless sensor networks due to diversity of possible applications. A priori performance analysis would thus reduce the design cycle for a WSN. The problem of systematic WSN engineering formulated in Deliverable D1.3 is as follows. Given a set of available communication components of different levels of modularity create a ranking system for their tuneable parameters which would bind the rules of their tuning and specific WSN application working in a particular communication environment. The ranking system will thus provide means to determine if a given protocol provides the required performance according specification of the particular WSN scenario.

In Deliverable D1.3 we described a methodology of ranking adjustable parameters of communication protocols on the performance in particular operating environment. We demonstrated the methodology by running a set of experiments with real hardware using simple topologies. We understood that a full-scale testing of the suggested methodology is possible only using computer simulations. Only in simulations one could experiment with a wide range of the topologies and environmental settings. We understood also however that current simulators are not ready for this kind of analysis. We develop this topic in this deliverable below.

3.2 Similitude of computer model of communication systems

Verification of the degree of satisfaction of the performance of communication system to the requirement of particular application could be done either by performing on-site experiments or simulations. The problem with on-site experiments with real hardware is that they are time and cost consuming, cannot easily cover the full range of expected operational conditions, and are very hard to repeat with the same settings. Simulations, therefore, are the only efficient means to evaluate the performance of wireless networks.

Similitude of a model is a concept used in to characterize similarity of different characteristics of a model to the actual modeled subject. Often this term is used in the context of modeling fluid mechanics, aerodynamics and marine engineering. The major problem associated with most current computer models of networks (network simulators), however, is that the implementation of network protocols in simulators is different from the one in real devices. Therefore, in many cases it is difficult to conclude whether a certain performance characteristics is due to specific protocol functionality or is just a feature of a particular implementation in a network simulator.

We want to apply this concept and come as close to reality as possible. Thus, eliminate additional steps in development process such as different implementations for simulations and real nodes. We want to use accurate models of communication channel which is provided in simulator without WSN support. Further, it is important to simulate realistic and scalable topologies and use diverse application in the same simulation, which is not possible today.

The last important objective for this work is to enable holistic simulations, where it is possible to test sensing algorithm interaction with communications process. In today's simulators the possible delays due to running algorithm for processing of sensory data on communications are often neglected or abstracted with On-Off type applications.

3.3 Background on TinyOS and NS-3

3.3.1 On TinyOS

With rapidly growing research in the WSN domain many operating systems (OS) were developed during the last few years. TinyOS [29][24][21], is de facto standard operating system in the WSN research community. Other operating systems for WSN are Contiki [25], MANTIS OS [22] just to name a few (for a full list see [31] and references therein). In some cases, more powerful wireless sensors use light-weighted linux-based OS. Since TinyOS was selected as the operating system for implementation of one of the demonstrators in the WSAN4CIP project below we summarize its major characteristics.

TinyOS, is an operating system designed specifically for wireless sensor nodes at the University of California, Berkeley. Below we list the highlights of TinyOS.

- TinyOS is a component based operating system that uses the event-driven design paradigm. In addition to this TinyOS implements a concurrency model which allows for two distinct execution modes: asynchronous and synchronous executions. In the synchronous mode a computational task once scheduled runs until the completion. In asynchronous mode a running task can be interrupted by an external HW interrupt. In this case the CPU resources are given to the interrupt handler code. Note that in TinyOS there is no dynamic context switching. This means that the programmer has to protect the critical variables manually ("atomic" declaration) when there is a risk of its modification during the asynchronous execution mode.
- TinyOS is de-facto standard operating system supplied with commonly used for WSN research Berkeley motes.
- A complete binary image of TinyOS kernel together with all applications is built during the compilation.
- When a sensor node needs another functionality, which is not present in the original image, another complete image should be downloaded to the node. Normally a sensor node keeps several binaries with different functionality stored in the re-writable flash memory.
- TinyOS specifies own extension to the standard C, called NesC. All applications are written in NesC. Upon compilation the NesC code first translates to ANSI C and the resulting intermediate file is compiled to the binary image.
- TinyOS is supplied with own simulation facility, named TOSSIM. It is a tool that is primarily used for debugging the TinyOS functionality. It may also be used for simple networking simulations. However, TOSSIM is not a general purpose network simulator, therefore, complex simulations with heterogeneous and sophisticated network settings and scenarios are not possible.

The major advantage of Tiny OS is the minimal code size amongst all three considered systems. The event-driven nature of the OS is proven to be efficient for a large class of WSN applications. In TinyOS a sensor node (device) is represented as a platform with a collection of hardware components. A TinyOS programmer thus wires together particular software and hardware components in a program image which implements the desired functionality. TinyOS is written in nesC, a dialect of C language. When the developer compiles the application, the nesC cross compiler will generate a platform specific C code. Further, the C code is compiled with the platform specific compiler to a binary image. This image is then loaded to the particular sensor platform for execution.

In order to enable the component-based design, TinyOS has a clear definition of Hardware Abstraction Architecture. This makes writing platform-independent applications and adding new hardware platforms a simpler task. The hardware is abstracted in three layers, namely: Presentation, Adaptation and Interface layers.

Hardware Presentation Layer (HPL) - components in this layer are directly above the hardware. They represent the capabilities of the hardware using the native concepts of the operating system. Components hide the implementation of hardware functionality providing an access to it with usual function calls.

Hardware Adaptation Layer (HAL) - the core of the architecture is build upon components from this layer. HAL components use raw interfaces provided by the HPL components and build abstractions hiding the complexity of using hardware resources. These components maintain the state that is used for performing arbitration and resource control. HAL interfaces expose specific features and provide the "best" possible abstraction that streamlines application development while maintaining effective use of resources.

Hardware Interface Layer (HIL) - components in this layer are the last building block of the TinyOS architecture. They take the platform-specific abstractions, which are provided by the HAL, and re-wire them to hardware-independent interfaces used by platform-independent applications. These interfaces provide a platform independent abstraction over the hardware. Having such abstraction, the application developer does not have to be concerned with specification of the hardware. HIL components export the typical hardware services that are required in a sensor node application. With clear hardware abstraction architecture, a sensor node is represented in TinyOS as a collection of components. Where the platform itself wires HAL and HIL layers with the HPL of the hardware. However, when matter comes to implementation of MAC layer functionality, for example, many radio transceivers implement MAC protocols on the driver layer. Therefore, even if algorithm is the same, the implementation of the MAC protocol is different between transceivers from different vendors.

3.3.2 On ns-3

Network simulator ns-3 is a successor of a well established in academic research community simulator ns-2. This is a discrete-event network simulator for Internet systems. Ns-3 improves the inflexibility of ns-2 by completely adhering to object oriented design paradigm by implementing whole functionality in C++. Ns-3 focuses on improving the core architecture, software integration and models of the ns-2 simulator.

The architecture of ns-3 separates the simulator implementation from the representation of devices, protocols, core, applications and common components. Thus, allowing both, simple extension to existing devices and protocols, as well allowing implementing real hardware and applications inside the simulator. Ns-3 has two simulation modes, real-time and simulator time, which opens new possibilities to perform realistic experiments. With the time abstraction the same simulation can be done both in real and simulated time. Thus, experiments could be conducted using both traffic patterns generated by real and simulated applications.

3.4 General idea behind our approach to design of a realistic simulation-framework for CIP WSN

We develop a new (virtual) node platform for TinyOS where the Hardware Interface Layer (HIL) is implemented in Linux with ns-3 bindings. Thus, we are keeping cross-platform compatibility of TinyOS applications. Therefore, applications, including an entire network stack (including MAC layer protocols), developed for our simulator can be run on real sensor nodes without any additional changes or adaptation procedures. This is on contrary to the existing implementation concept where the MAC layer functionality is implemented in hardware drivers. The role of ns-3 is to provide access to the simulation engine (scheduler, timers) and implementation of radio channel models. Ns-3 will also use to configure and run actual (large scale) experiments. Inside ns-3 each node running TinyOS image is created as a separate object each with its own memory space. This will allow us to experiment with different applications in the same simulation.

When integrating TinyOS with ns-3 a number of challenges have to be addressed. Among the most difficult ones are to correctly replicate radio devices available for sensor nodes; to reuse tracing functionality from ns-3 inside the TinyOS and finally to integrate the TinyOS and ns-3 components.

In Figure 7, the layout of our framework is presented. On the left part of the figure we have the actual application, which uses the network stack and the hardware. When compiling the TOS image on the node, for a specific platform, the compiler will use the defined HPL and HAL wiring to produce the executable code. The box with a bold text “ns3 hardware driver virtualisation” indicates the place in the TinyOS architecture where the bridging with ns-3 is done. This integration is no different from the current approach to implementing support for new hardware platforms. However, in some cases, developers of drivers for the new platforms choose to implement either partially or completely the functionality of MAC protocol in the radio driver part. In our implementation, the MAC functionality is a software component in TinyOS, only the radio device drivers are virtualized.

This approach is drastically different from the approach taken in the TOSSIM simulator. Firstly TOSSIM abstracts the MAC layer away by own implementation different from the actual functionality of the protocol. Secondly TOSSIM allows creating only one instance of the TinyOS image and the specific node is represented as entry with the state of variables in hash table which limits the number of different application in the experiment to one.

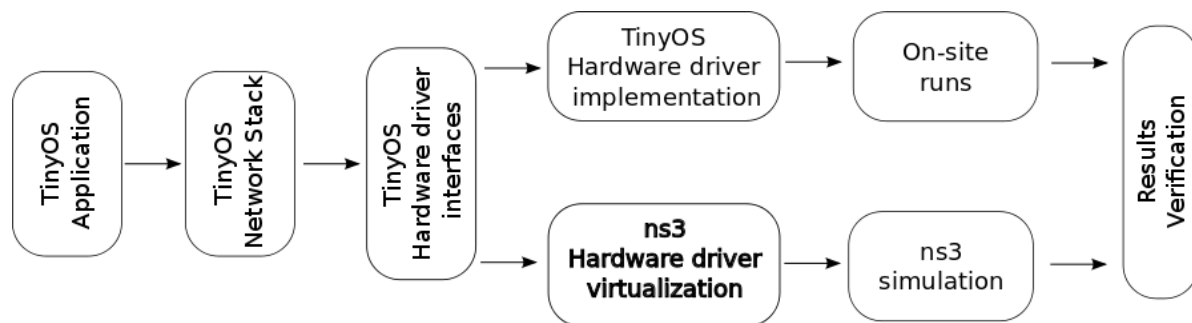


Figure 7: In bold is shown the system part where the tos-ns implemented.

3.4.1 System design

The key challenge when virtualizing TinyOS in NS-3 is how to access and pass the information between the two frameworks. The essential for TinyOS information includes getting time, accessing channel. Further, ns-3 simulator will need to notify the TinyOS about activities on the channel and timer events. Therefore, we need to establish bi-direction communications between the TinyOS node image and the ns-3 environment. We want to provide I/O interactions for the sensing capabilities, which is done by exporting HIL layer of I/O ports.

Main components that we virtualize as part of ns3 platform in TinyOS are listed below:

- MainP - components takes care of initializing the system.
- HilTimer - components provides access to ns-3 timer.
- Transceiver - components provides bidirectional interface for the virtualized transceiver device.
- I/O - component that provides virtualisation of the sensing device.

Further, we develop a set of new classes in ns-3, which provide the needed functionalities in TinyOS-ns3 platform. These classes are listed below.

- RadioTransceiver-class represents the transceiver device, and provides bidirectional interaction between network components of TinyOS and ns-3 channel.
- TosNode - class that holds necessary for ns-3 information about the state of the node. The information may include, but is not limited to, node position, its neighbors, node state (on, off) etc.
- GlueClass - this class hides the implementation of bidirectional communications between TinyOS image and TosNode object.

Figure 8 describes the interaction between different parts of the framework. On the left side the TinyOS environment is shown where we abstract parts of hardware and connect it to the TOS-NS environment. The TOS-NS environment serves as glue between TinyOS and ns-3 simulator environment. TOS-NS provides also means for interaction with other simulators, for example a simulator of dynamic models. This interaction provides a unique opportunity to test the effect of sensing algorithms on the communication process.

The build process starts by compiling TinyOS application to a library, further it links and compiles the ns-3 environment with TinyOS.

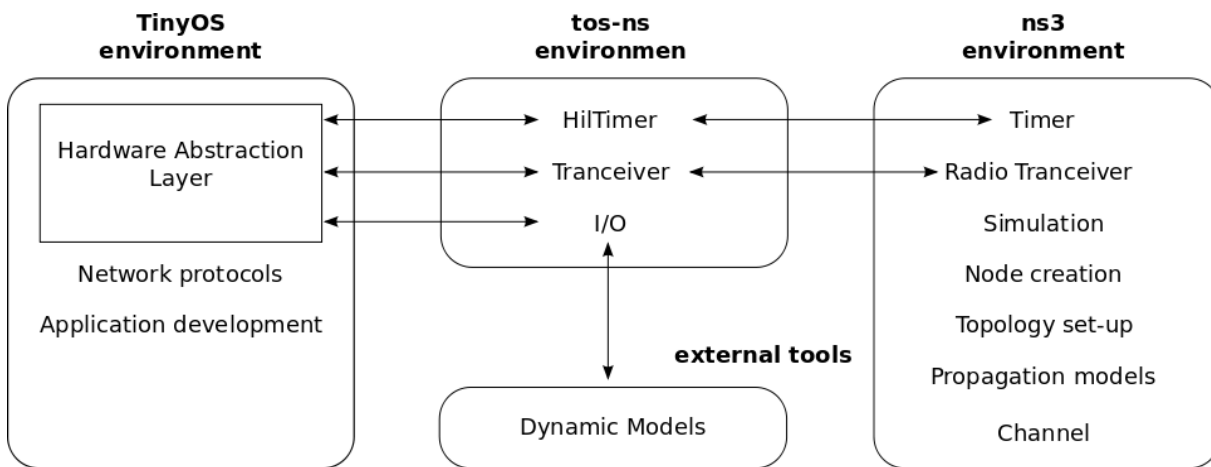


Figure 8: Dependencies between TinyOS, ns3 and the role of TOS-NS.

3.4.2 Implementation

There are several ways to implement the desired framework. For instance we could convert TinyOS generated C code to a C++ class, however this would change the code base of the image. We could create a library and load its multiple instances one for each node. This is a programming challenging task since we need bi-directional communication. On option to organize communication between the TinyOS and ns-3 parts of the framework is to run through sockets. This, however, will limit the number of supported nodes in one simulation to 16000 due to the limit on the maximum number of sockets. Figure 9 shows our approach to implementation of the described framework.

We create a new (virtual) hardware platform in TinyOS. This platform represents and uses desired functionality from ns-3. Within the ns-3 framework we develop virtualized hardware drivers, which become part of ns-3 simulator. Under the TOS-NS environment we post-process the generated by the nesC compiler C file in order to make it C++ compatible and create a binary library, containing implementation of a sensor node. Using ideas from external polymorphism pattern [23] we glue the TosNode representation in ns-3 with the library created from TinyOS code. Thus, each TosNode loads its own library in separate memory space. TosNode implements the previously described functionality of the virtualized hardware. We use ns-3 to create the simulation itself, including topology setup, physical channel and logging.

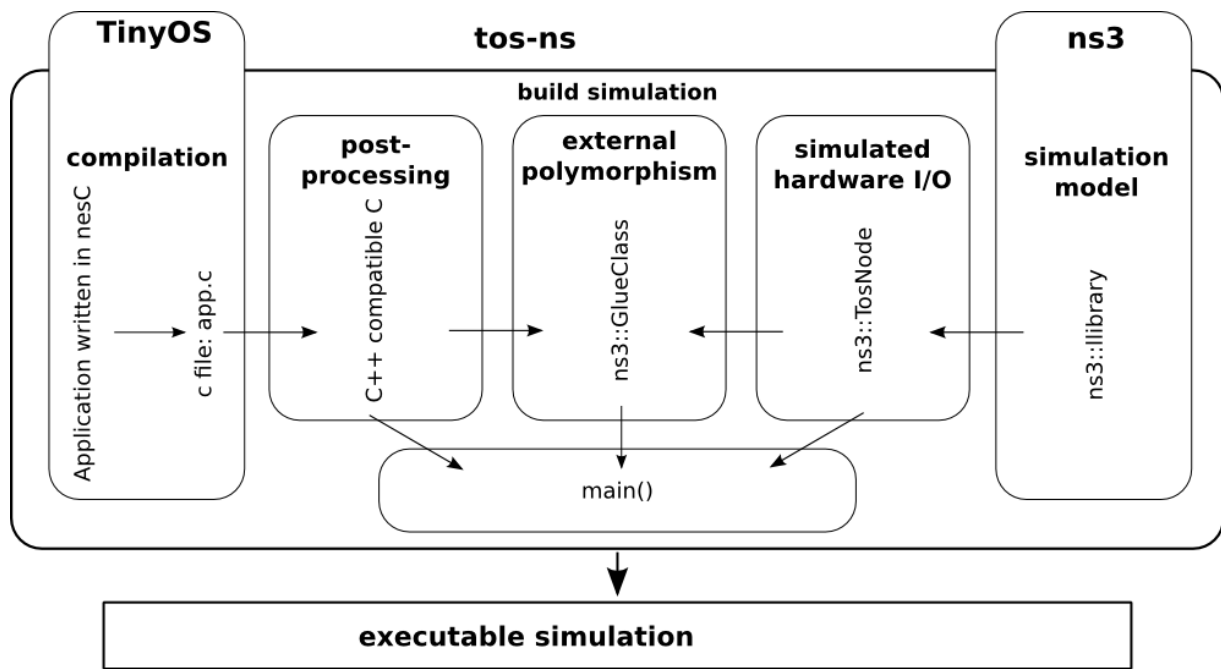


Figure 9: TOS-NS implementation and compilation component flow

3.5 Summary

To the time of writing this document we obtained the first version of the combined TOS-NS environment and currently finalizing the implementation. Due to the complexity of the implementation process we expect that technical problems will appear during the first round of experiments, therefore the work on framework finalisation will continue beyond the scope for WP1. In particular we will implement and verify in our simulation framework a modularized MAC protocols as an activity in WP3.

4 Formal WSAN Description Language

In this section we present a detailed WSAN description language that maps the properties and databases introduced in Section 2 on a system-independent XML scheme. We are convinced that a reasonable and accessible knowledge management is one essential key for an efficient engineering process. The description language presented here is a second step toward such a powerful knowledge management. We build on the XML description language developed in the UbiSec&Sens project and refine it with several key elements.

4.1 XML System Description Language of UbiSec&Sens

4.1.1 Introduction

This section presents the XML system description language developed in the context of the project UbiSec&Sens [20], FP6-2004-IST-4, contract no. 268206FP, which preceded the current WSAN4CIP project. The description presented here was already modified to exclude specific references to the UbiSec&Sens project and simple adaptations were done regarding the WSAN4CIP. In section 4.2 extensions are added to tackle specific requirements of the WSAN4CIP project.

The main objective of the system description language is to be able to express all the relevant aspects of a given WSAN, both hardware and software. The description uses the Extensible Markup Language (XML) [7]. This standard was chosen due to its flexibility, expansibility and broad dissemination, with plenty of tools available.

The XML schema allows the description of physical characteristics of wireless sensor nodes. It lets specify types of nodes and detail their constitution, namely, the processor used, power consumption, memory available (type and size), sensors and actuators available and communication facilities.

With this information, and based on a set of requirements for a given application, it will be possible to select the nodes that best satisfy the needs, and specify the concrete composition of the WSAN to be deployed.

The schema allows also describing software components (modules) and their interfaces. Based on this information, a pool of software components can be developed and made available for future selection. The information in the XML schema support the specification of different aspects of the software modules, such as, its memory footprint, estimated energy consumption, security level and interfaces provided.

The available information supports the selection of modules that satisfy the requirements of a particular application and allows validating if the selected modules have compatible interfaces and, for example, if they have the adequate level of security. Furthermore, it is possible to identify the various types and amounts of memory required and help select the appropriate node's hardware to deploy the application.

The schema defined is flexible and most of the information is optional. This supports an incremental description process where information can be added when it becomes available or when it is required. The more complete the information, the more useful the specification can be. Another important aspect is that, given the intrinsic properties of XML, the specification can be easily expanded in the future to fulfil new requirements.

4.1.2 Schema Overview

Figure 10 presents an overview of the XML schema. As illustrated, a WSAN will be identified by a name, will have a deployment version and, optionally, a description.

A WSAN is constituted by any number of nodes (NodesList). To allow the characterisation of the nodes and their sensors and actuators, several "types" are used: SensorType, ActuatorType, ProcessorType, MemoryType, CommunicationType, SoftwareInterfaceType, SecurityType, FunctionalGroup and SoftwareComponentType. The elements shown in the Figure correspond to lists with any number of the above types. As the names suggest, these types describe the characteristics of the available sensors, actuators, processors, memories and communication hardware used in the nodes. The SecurityType will be used to define security characteristics for the software modules. The element FunctionalGroup allows defining functional groups

and sub-groups that will be used to classify the software components and ease the selection process of components, accordingly to its functionality.

The elements SoftwareComponentType and SoftwareInterfaceType describe the software modules and their interfaces.

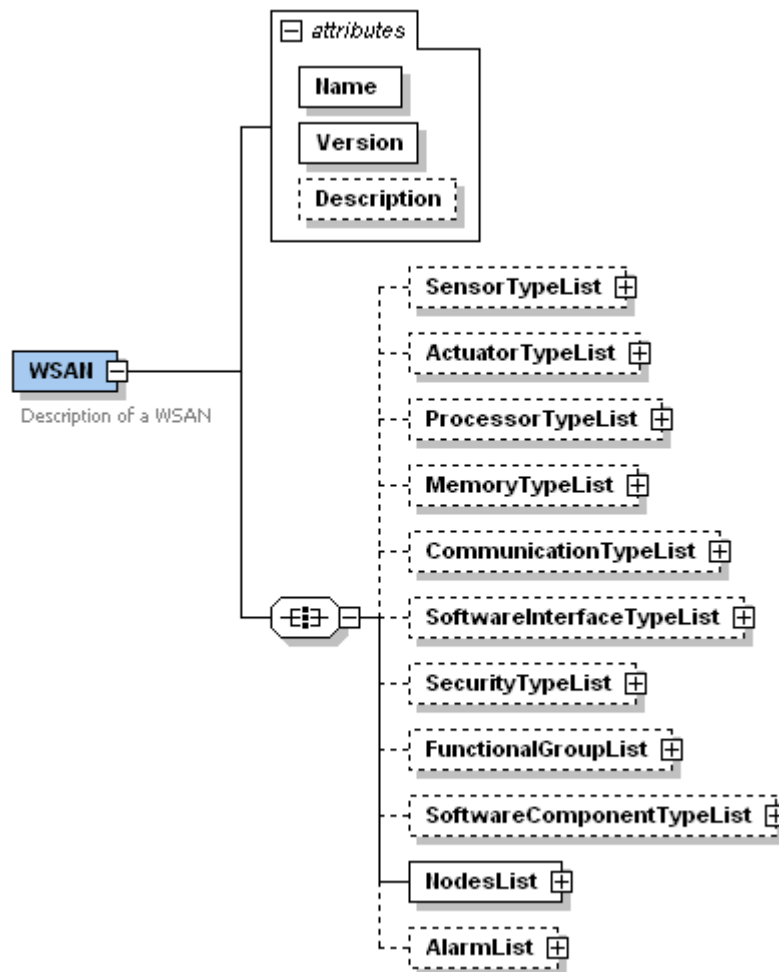


Figure 10: Description of a WSAII

In the following sections the mentioned types will be described and it will be detailed how each node’s characteristics are specified. A node specification details both its hardware and concrete software modules deployed on it.

4.1.3 Hardware Properties

Hardware properties are described in the SensorType parameter. The data type describes the data representation of the values read from the sensor and can be, for example, Boolean, Int16, UInt8. The attribute “units” refers to the units of the physical entity measured. It can be, for example, Celsius Degree, Watt, RH – Relative Humidity, Lux.

For each sensor type one can specify (optionally) the following characteristics: a “TimeToSample” (time needed to read a value), an EnergyConsumptionPerSample (energy required to execute a reading) and a sampling rate. Each one of these elements may have an attribute “unit” to specify the units of the correspondent value.

Similar representation is used for the actuators. Each element `ActuatorType` describes the data representation of the values sent to the actuator, such as, Boolean, `UInt8` or `Int16`. For each actuator type one can specify (optionally) the value of energy consumption associated with an actuation. The energy unit used can also be specified.

A dedicated structure for processor type one can specify (optionally) the following characteristics: energy consumption, operating voltage, word size (e.g., 8 for an 8-bit processor or 16 for a 16-bit processor) and clock rate. It is even possible to specify different values of power consumption in accordance with the processor operating mode (e.g., normal mode, standby, sleep).

A WSN node can have different types of memory which can be used for storing program code, temporary data, configuration data or data read from sensors. These types of memory can be built in the processor or use external integrated circuits.

The XML schema allows the definition of different types of memory. Each one will have an identification (`MemoryTypeId`), a name (or model reference), a type (e.g., RAM, FLASH, EEPROM) and a description (optional).

For each memory one can specify (optionally) the following characteristics: energy consumption (for various possible modes of operation), operating voltage, word size (e.g., 8 bit, 16 bit), read access time (may have different values for each operating mode) and write access time (allows also various values).

A WSN node is typically equipped with a radio transmitter and receiver. However, a node may also have other means of communication. That is what happens with sink nodes that, commonly, interface with a PC using a serial communication line or a USB connection. It is also common sink nodes that interface directly with an Ethernet network.

The element `CommunicationTypeList` allows the definition of any number of wireless and wired communication facilities. For each communication device one can specify (optionally) the following characteristics: energy consumption for different operating modes (e.g., receiving, transmitting, standby), operating voltage, bit rate (e.g., 250 Kbps), and a frequency band (e.g., 2.4 GHz).

We list the properties here because - even though we decided not to use explicit hardware descriptions – the properties will be part of the new more flexible component description. Instead of explicit hardware descriptions, we have general components with properties that describe the hardware. Thus hardware components are treated like software components – they just have a different, more general, syntax.

4.1.4 Security Aspects

Security is a very important aspect for project UbiSens&Sec. Different software modules are being developed with various security characteristics. To support the process of module selection based on their security features, the XML schema has an element (see Figure 11) that allows the definition of any number of security types.

Each security type has an identification (`SecurityTypeId`), a name, a security level (small integer that indicates a generic degree of security) and a description (optional).

Furthermore, a security type may be characterised using any number of security properties.

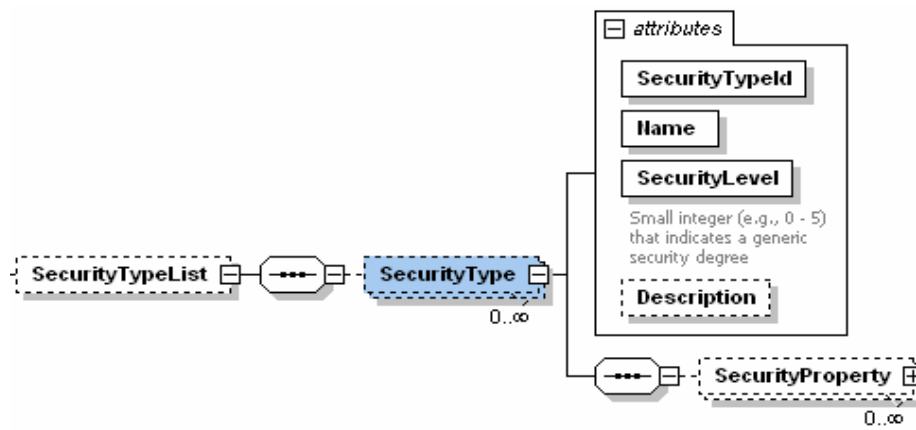


Figure 11: Security characterisation

Each security property (see Figure 12) has the following attributes: a name (e.g., Integrity, Concealment, Robustness), a generic security level (optional) and a description (optional).

A security property can be characterized by any number of security parameters. Each one is constituted by pairs <name, value> describing a specific characteristic (for example, an ECC algorithm using 233 bits).

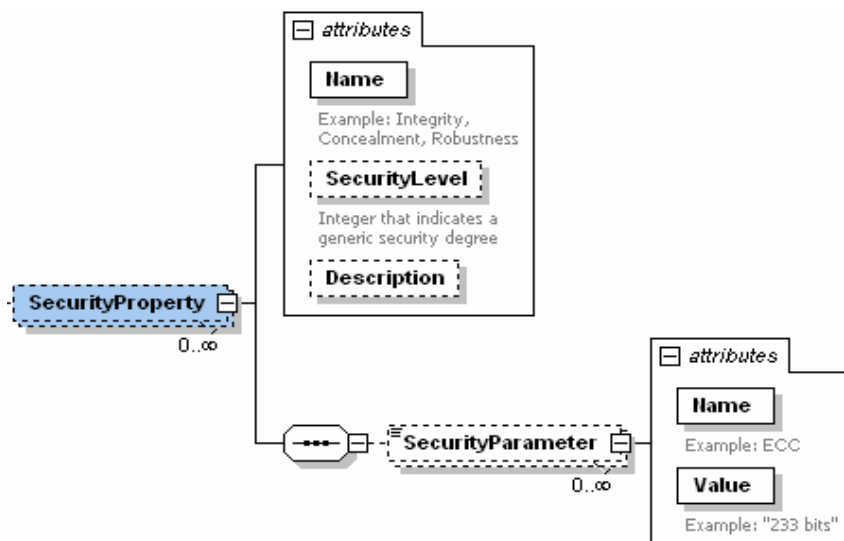


Figure 12: Security parameters

Similar to the hardware properties, the security parameters should be part of a more general property description. A weakness of the UbiSec&Sens description is that small changes in parameter structure (for example a new security parameter) would need the whole XML structure to be adapted.

4.1.5 Modules Functional Grouping

The XML schema supports the definition of functional groups which can be used to classify software components and ease the process of their selection. Each group can be divided into sub-groups. For example, it is possible to create the group “Cipher” and divide it into “Symmetric”, “RSA” and “ECC”.

Both functional groups and sub-groups have the following attributes: an identification (FuncionalGroupId), a name and a description (optional).

As can be seen, it is proposed that groups and sub-groups share the same identification space, which will ease their reference. This can be particularly relevant as each software component may belong to various groups and sub-groups.

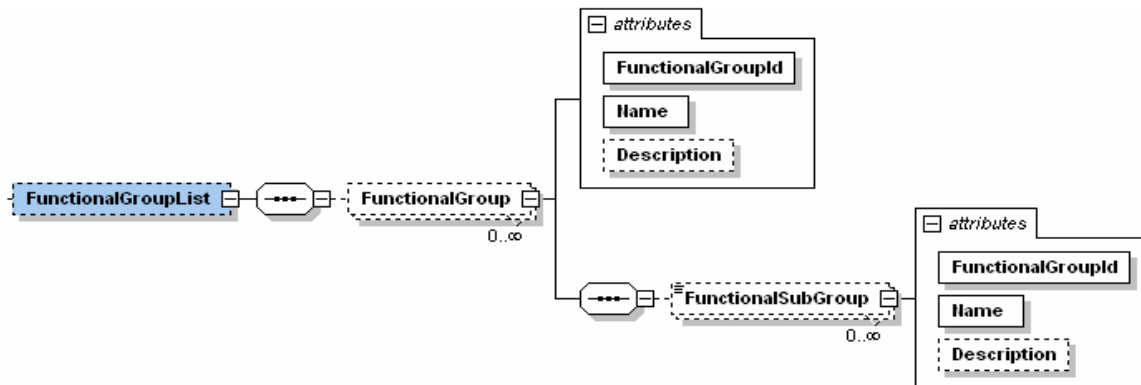


Figure 13: Functional grouping

We expect to use that parameter in a similar way in our description.

4.1.6 Software Description

4.1.6.1 Software Interfaces

Software components interact with each other using interfaces. Each interface type (see Figure 14) has an identification (SoftwareInterfaceTypeId), a name, a version (optional) and a description (also optional).

Interfaces are bidirectional supporting the calling of commands (in one direction) and a call-back mechanism (in the opposite direction) for event notification. This bidirectional characteristic supports directly a TinyOS/nesc feature. However, commands and events are optional, allowing also the specification of unidirectional interfaces.

Each interface may have any number of commands and events, which are described by their signatures.

A command signature has a name, a description (optional) and a return data type. Each command signature may have any number of parameters, each with the following attributes: a name, a description (optional), a data type and an indication of the parameter passage convention used (by value or by reference).

Event signatures have the same characteristics as command signatures.

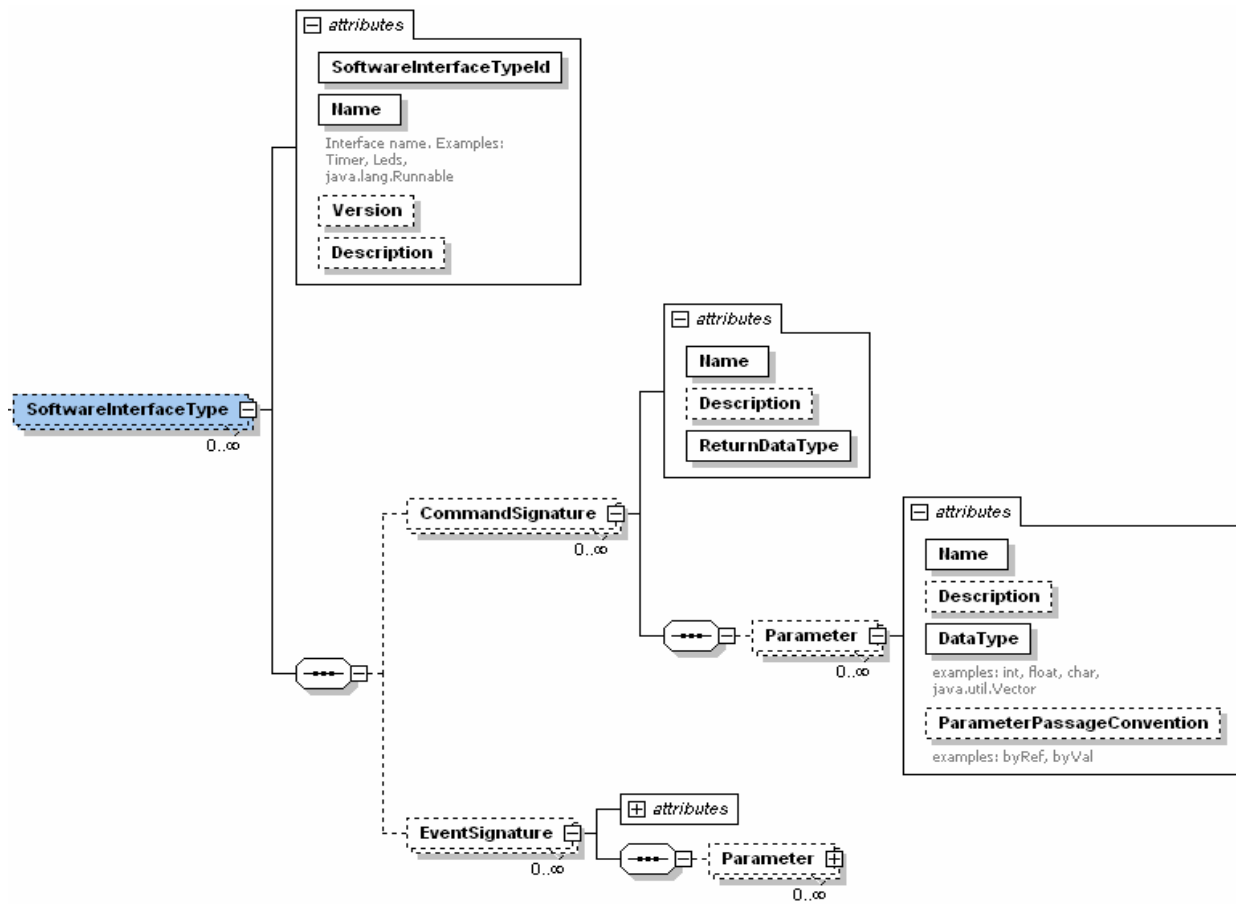


Figure 14: Software interfaces

4.1.6.2 Software Components

The UbiSec&Sens XML schema allows the specification of any number of software component types. These component types can, later on, be instantiated and deployed in the sensor nodes.

Each software component type (see Figure 15) has the following attributes: an identification (SoftwareComponentTypeId), a name, a reference to a security type (that details the security characteristics of the component), a version, a classification of the role of the component (e.g., Operating System, Protocol, Application) and a description.

Most of these attributes are optional, with the exception of the identification and name.

Furthermore, for each software component type, one can (optionally) specify: an energy consumption, a code memory size, a data memory size, a persistent memory size (typically for data acquisition), any number of functional groups and sub-groups to which the component belongs and the software interfaces it provides (for use by other components) and uses (implemented by other components).

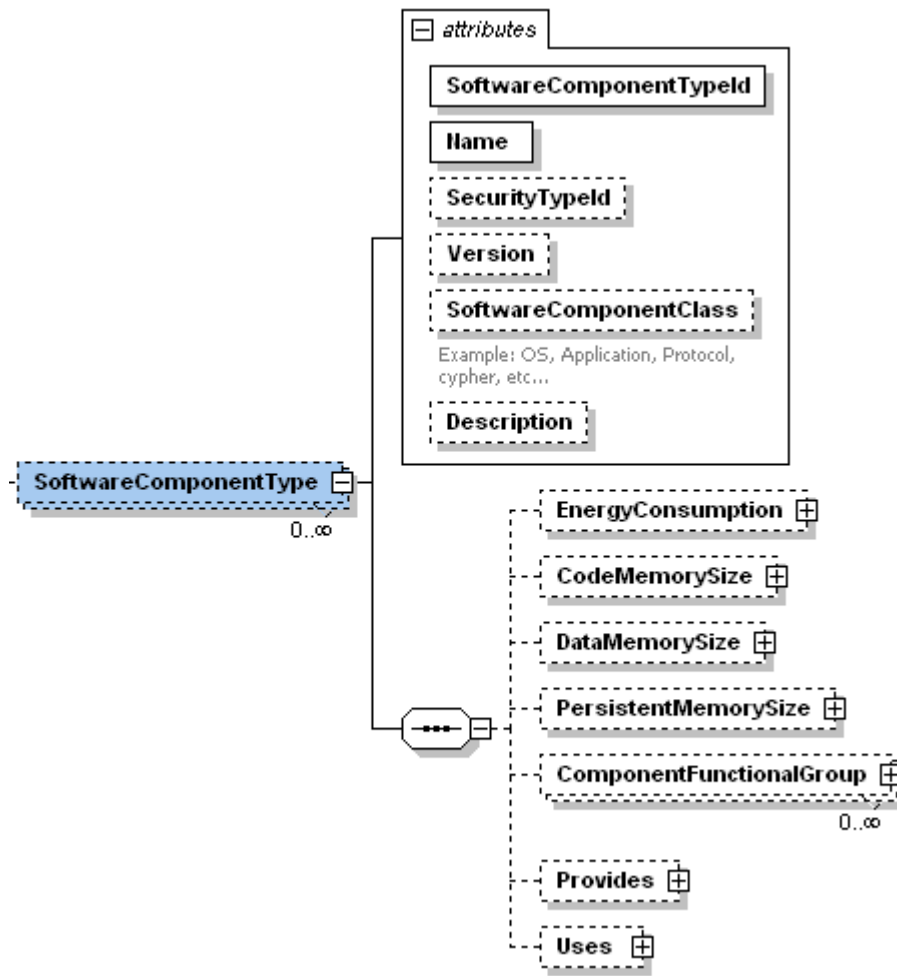


Figure 15: Software components

As illustrated in Figure 16, each software interface used or provided by a component has a reference (SoftwareInterfaceTypeId) to the corresponding interface type (where the interface is described). Optionally, it is possible to give an alias to the interface name. This is a feature used in the TinyOS/nesC development environment that is supported by the XML schema.



Figure 16: Software components interfaces

4.1.7 Nodes description

Each node of the WSN will have an identification (NodeId) and a name. And each node may be characterised in terms of its hardware and software composition.

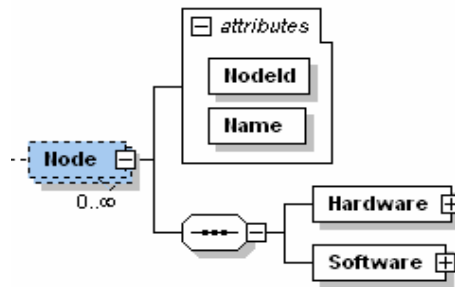


Figure 17: Description of a node

4.1.7.1 Node Hardware

The hardware description of a node is illustrated in Figure 18.

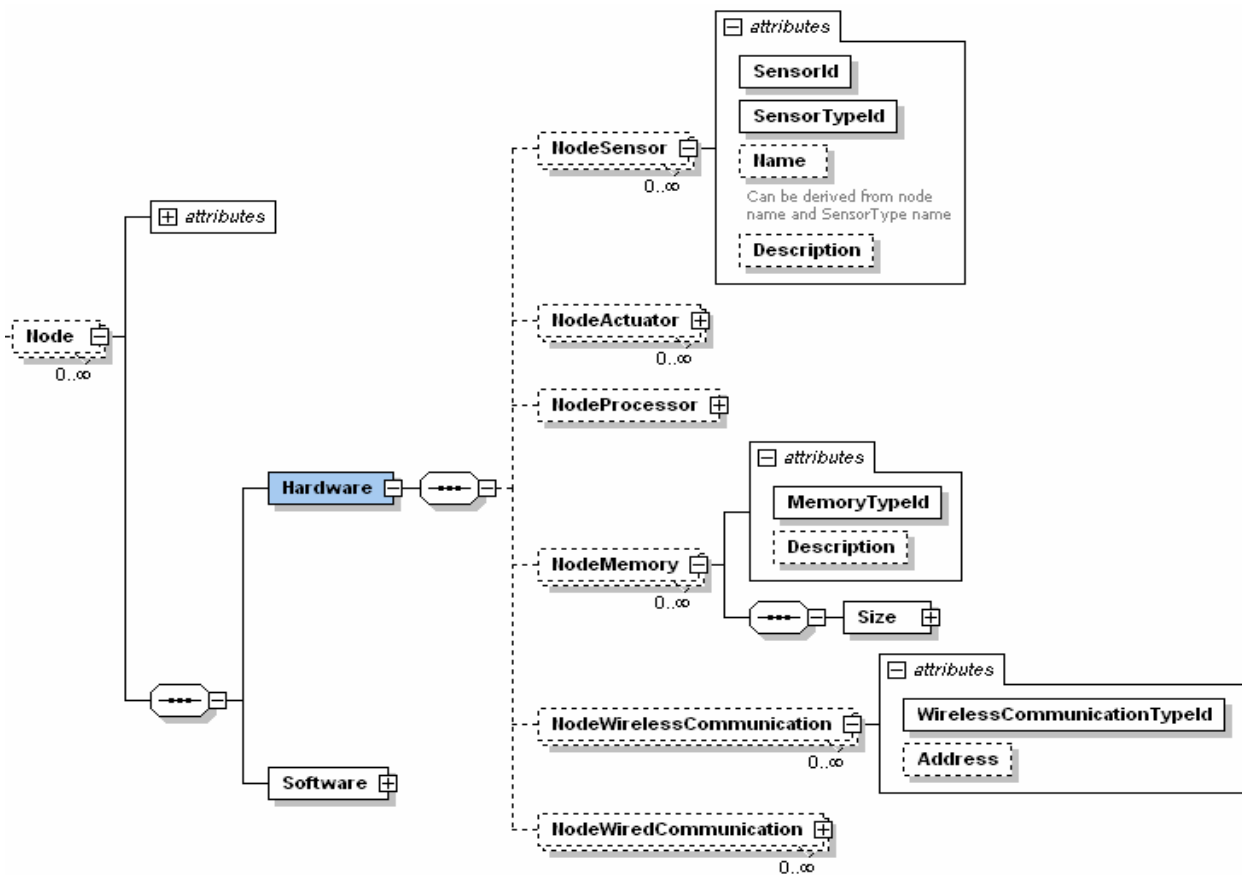


Figure 18: Node hardware

For each node one can, optionally, specify the following characteristics: any number of sensors and actuators, the processor used, the type and size of memories available and the existing communications facilities.

The characteristics mentioned are instances of the corresponding item types, which were described previously. As such, each element contains a reference to the corresponding element types and attributes that are specific of the instance. For example, for each sensor in a node, there will be a concrete sensor identification (SensorId), a reference to its type (SensorTypeId) where its characteristics are described and other specific data such as a name (optional) or a description (also optional) – see Figure 18. The same happens with the actuators.

The node’s processor makes reference to its type and may include a description.

The memory available in a node references memory types defined previously, may include a description and specifies the correspondent memory sizes.

The available communication infrastructures can be described making reference to types defined previously and, optionally, specifying the address associated with each communication interface.

4.1.7.2 Node Software

The software description specifies concrete software components deployed in a node – see Figure 19. These components have a concrete identification (SoftwareComponentId) and make reference to types defined previously (SoftwareComponentTypeId). It is possible to specify names for the components instances to differentiate them.

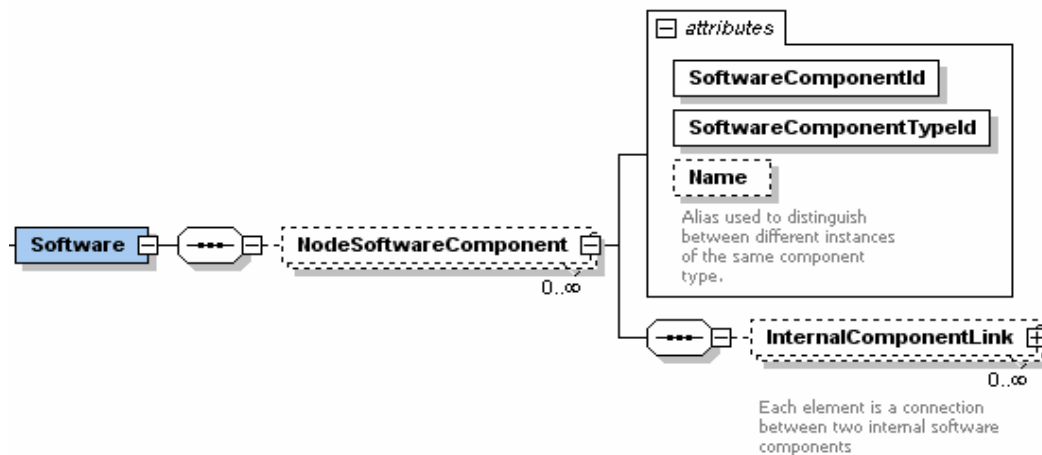


Figure 19: Node software

A software component can be constituted by other components. In this case, it is possible to specify how internal components are interconnected (linked). To this end, one can have any number of elements that specify those interconnections (InternalComponentLink).

To better understand this feature, which is supported by the TinyOS/nesC development environment, we illustrate in Figure 20 the relevant concepts. A component interconnects to other component using a software interface, which is the linking element. The specification of this linking implies the identification of the software interface (SoftwareInterfaceTypeId) and the components involved (LinkedFromComponentId and LinkedToComponentId).

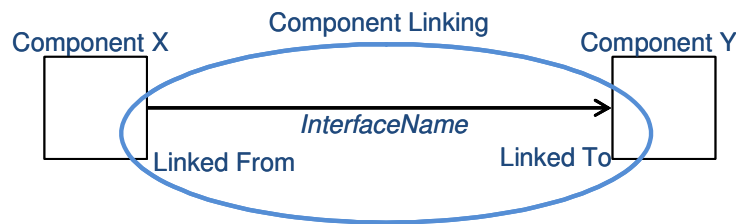


Figure 20: Interconnection of software components

The principles of this software and hardware description will be applied in the new description language as it will be explained in the following sub section.

4.2 Extension of the XML language to support description of communication modules

This section presents an extension to the XML description language to support the specification and selection process of communication modules, in the context of the WSAN4CIP project. The main objective is to have a flexible way to represent the relevant characteristics of each software component in order to be able to assess their compatibility and fulfilment of requirements defined by the user, accordingly with the methodology explained earlier in this deliverable.

The XML defined uses the *extension* feature of XSD. This is a powerful and flexible capability as it allows defining an element and then reuse and expand it whenever necessary.

4.2.1 Parameter Type

A novel concept of the WSAN description language is the general parameter type which will be used for many

Since all requirements, properties and parameters are basically the same type, they can operate on each other.

ParamType, as shown in Figure 21 is a complex type that contains some static information (id, domain, name) but also a complex type optionally providing either a choice, a value range, a formula or a fixed value.

- **Level** is the possibility to select between a list of possible selection. Thus it is used for nominal metrics.
- **Range** defines a range [min, max] for a value. It represents a range of floating point numbers with a given resolution. If the resolution is 1, it represents a range of integers. Thus it will be used for interval metrics.
- **Formula** defines a formula that computes a value from other variables or params. It is similar to the formulas that can be entered in the cells of today's spreadsheet program (=A1+B3)
- **Fixed** is a fixed value. A fixed value can be part of the range or of the formula, but to improve understandability of the data structure we decided for a dedicated item.

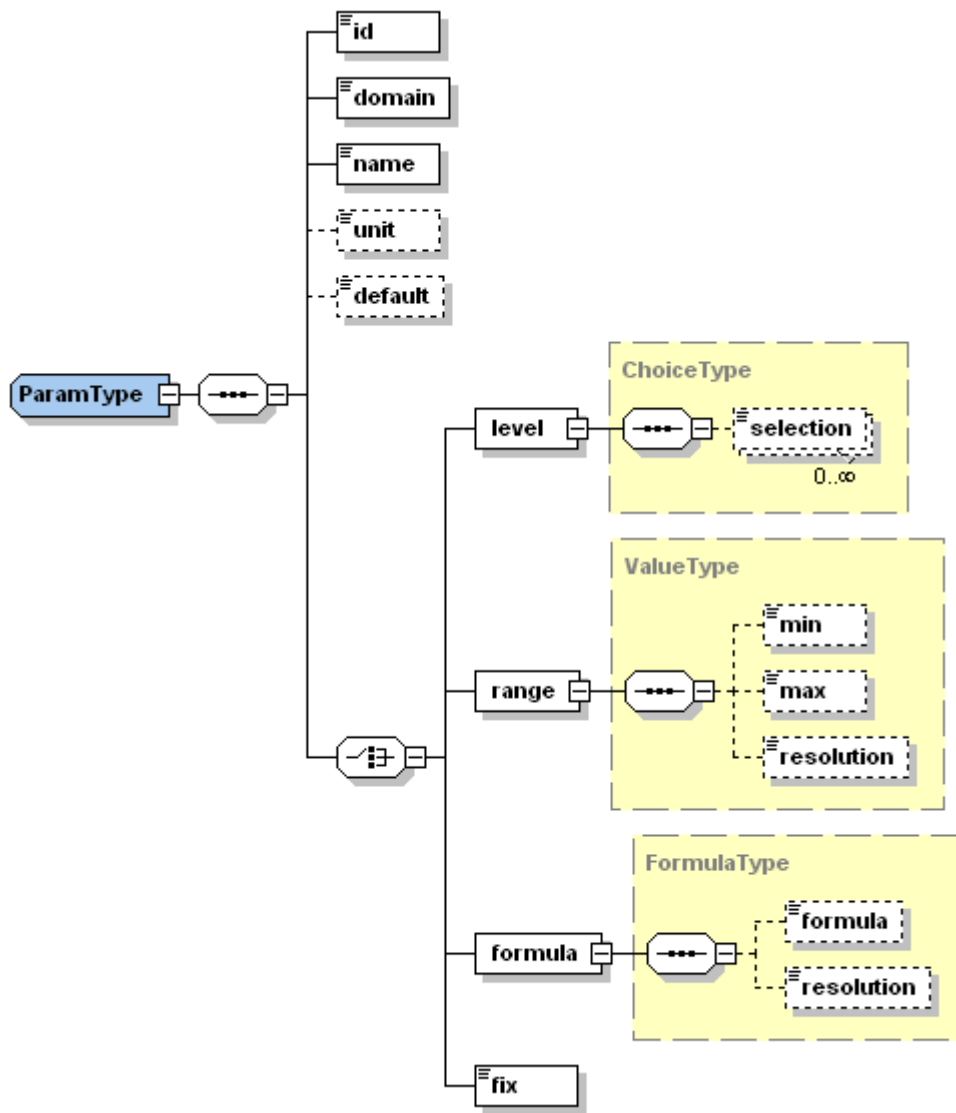


Figure 21: Definition of a parameter type

Figure 22 shows how the ParamType struct inherits its properties to structs of other data types.

Basically we have two sorts of property types: templates and instances. Templates are part of the databases. They usually describe a range or a selection for the specific property. For radio module such a template selection could be the applied modulation. In that case the selection could be “FSK”, “PSK” or “QPSK”.

The actual instance defines the parameter for a chosen and parameterised module. Then for the component instance one modulation is chosen. Syntactically it is very similar to the template – but the meaning is very different.

Figure 22 shows eight derived types from the ParamType:

- requirement and requirementSelection represent requirements in the database (as template) and as actually chosen requirement for the application (selected instance)
- component.use.param is an interface parameterisation as we will describe in the following section. It can be used either as template inside the component database, or an instance, when the module is chosen and the parameters are set.
- Component parameters are parameters of the components. Their meaning is similar to the interface parameters.

- Component properties are mostly formulas or fixed values. In case of a formula the instance will not be different to the template. But we keep the concept due to homogeneity of the data types and for potential future use.

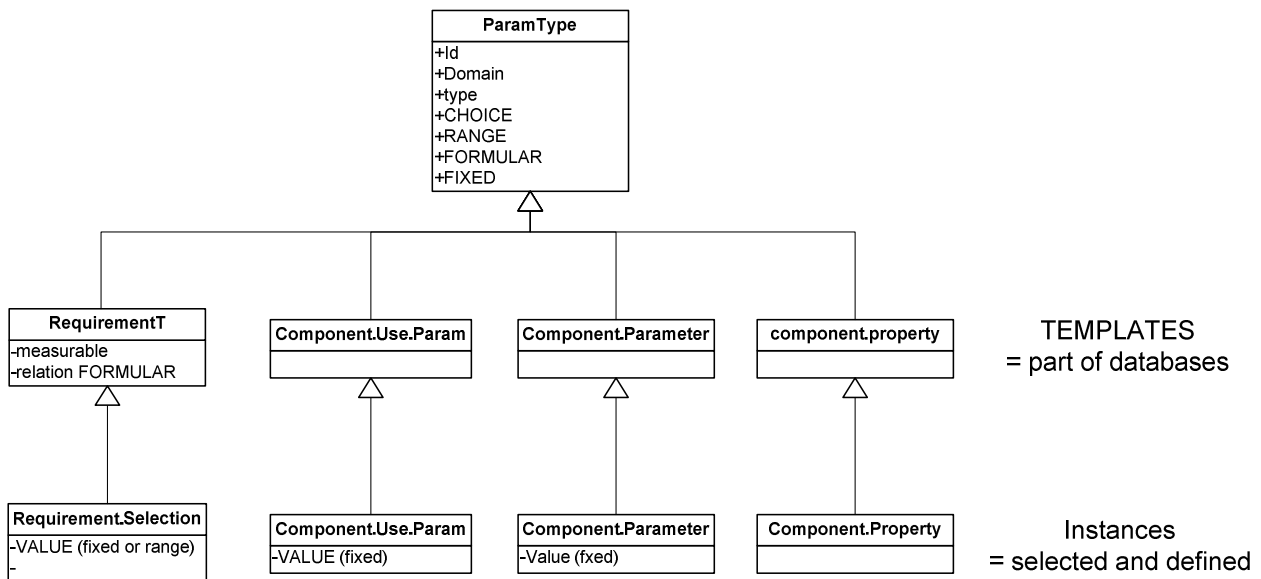


Figure 22: Parameter structure: parameters, requirements and properties inherit from the Param-Type and only add few individual types.

That requirements, component parameters and properties are basically the same data type allows combining and comparing them in all steps of the development process. For example requirements can be compared to properties. Even the realisation of a strict type checking is feasible using this language construct.

By this the ParamType and the derived data types can replace most fixed XML data types as they were used the UbiSec&Sens description language in a very flexible and reusable fashion.

4.2.2 Component Description

We start by presenting the base elements which are illustrated in Figure 23. The figure shows the definition of a ComponentType which will be used as the base for the definition of software components (also designated as modules).

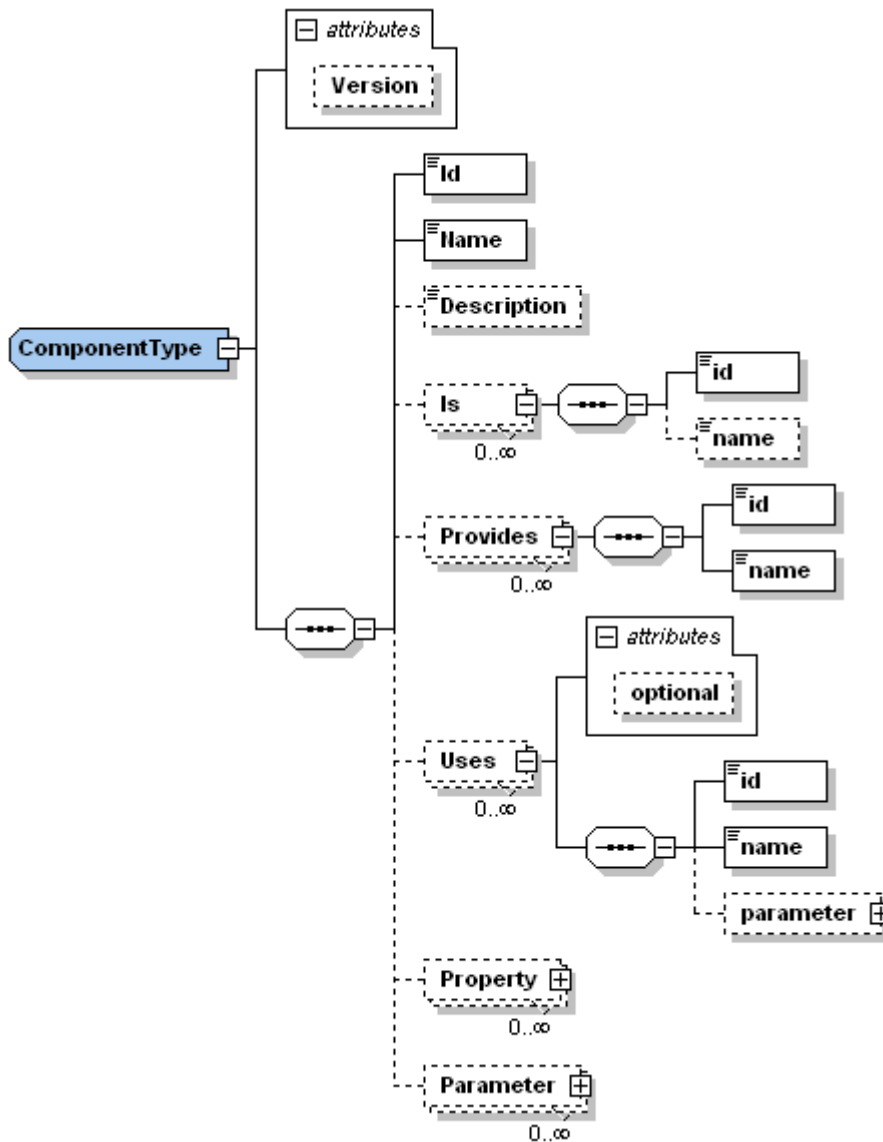


Figure 23: Definition of a component type

Each Component Type has an “Id” and a “Name”, and may include several optional elements: “Description”, “Is”, “Provides”, “Uses”, “Property” and “Parameter”. “Description” is auto-explicative, allowing the description of the component type. “Is” is used to classify functionally the component type. This element may appear in any number allowing a component to belong to multiple functional classes.

“Provides” and “Uses” may also appear in any number and are used to define the interaction with other components. “Provides” refers to services provided to other components and “Uses” refers to services required from other components. Please note that “Uses” has a Boolean attribute “optional” that may be used to indicate that the a use a certain service is optional.

“Property” and “Parameter” may appear in any number and are used to characterise properties of the component and configuration (static) parameters. These are both of type ParamType described in Figure 21.

Figure 24 and Figure 25 define, respectively, a software component and an interface. They are both defined based on the “ComponentType” described in Figure 23.

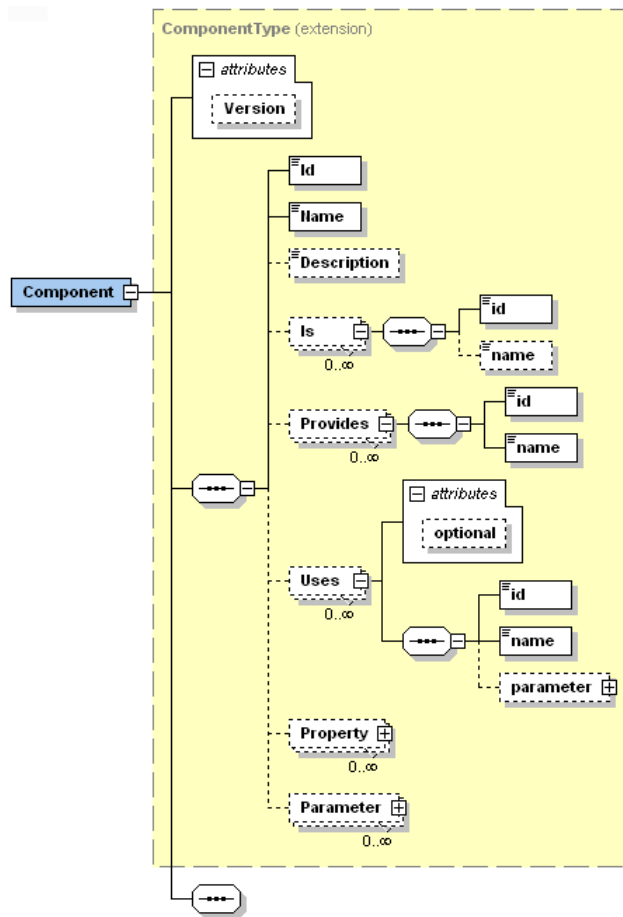


Figure 24: Definition of a software component

The definition of an interface is based on the “ComponentType” to simplify the specification. Although the component type can express more information than required for an interface, it includes the relevant data and this avoids the definition of a new element.

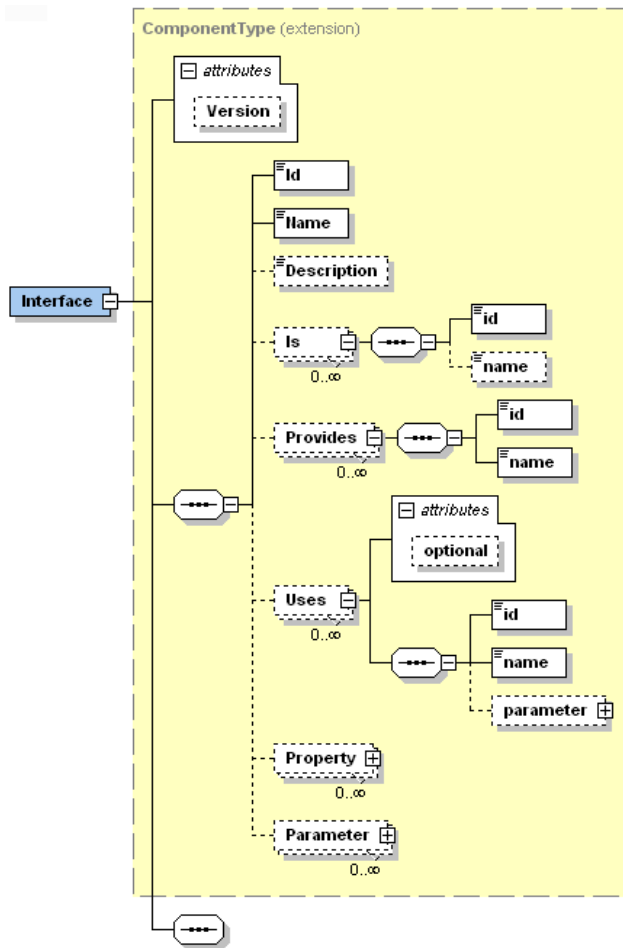


Figure 25: Definition of an interface

4.2.3 Requirement Description

Finally, Figure 26 illustrates the definition of a set of requirements. These are specified by the user and are

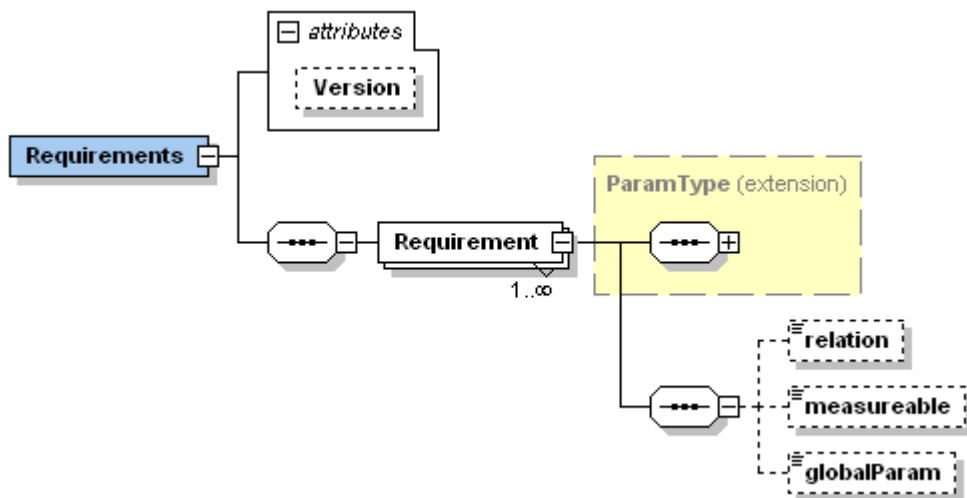


Figure 26: Definition of requirements

defined based on a parameter type (“ParamType”) which was described earlier. This parameter was extended to include, optionally, three elements: “relation”, “measurable” and “globalParam”.

Relation is the formula that describes the relation to other properties. As part of the system test the relation must be valid in order to pass the test. The relation can be an assignment (e.g. Frequency:=1/Period) or a comparison (HbH_distance<=EtE_distance/nodes) or a list of sub-formulas, similar to program code.

Measurable and globalParam are two booleans to indicate the sort of the requirements: globalPArams are global constraints, while measurable requirements can be computed or simulated and have to be compared in the final testing.

4.3 Examples

As an examples this subsection shows the component description of the CC1100 radio module as it is part of the component database. Comments (cursive) explain the blocks.

header

```
<?xml version="1.0" encoding="UTF-8"?>
<Component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="w4c.xsd" Version="1.0">
```

```
<Id>M018</Id>
<Name>CC1100</Name>
<Description>Transceiver TI</Description>
```

It provides the radio interface. The interface has the id I006.

```
<Provides>
  <id>I006</id>
  <name>Radio</name>
</Provides>
```

Property current consumption of the transmission as formula based on data in the technical datasheet [33]. The formula uses the ? : operators as it is known from many programming languages.

```
<Property>
  <id>Pr0001</id>
  <domain>energy</domain>
  <name>TX current consumption</name>
  <unit>mA</unit>
  <formula>
    <formula>
      band=315?TX power= -10 ? 27 :
        TX power= 0 ? 14.8 :
        TX power= -6 ? 12.3 : 0 :
      band=433?TX power= -10 ? 28.9 :
        TX power= 0 ? 16.9 :
        TX power= -6 ? 13.5 : 0 :
      band=868?TX power= -10 ? 31.1 :
        TX power= 0 ? 16.9 :
        TX power= -6 ? 13.5 : 0 : 0
    </formula>
  </formula>
</Property>
```

The voltage property is fixed to 3 volt

```
<Property>
  <id>Pr002</id>
  <domain>energy</domain>
```

```

<name>voltage</name>
<unit>V</unit>
<fix>3.0</fix>
</Property>

```

The data rate parameter is a choice out of 3 data rates

```

<Parameter>
  <id>P00 </id>
  <domain>radio</domain>
  <name>data rate</name>
  <unit>baud</unit>
  <default>38400</default>

  <level>
    <selection>1200</selection>
    <selection>38400</selection>
    <selection>250000</selection>
  </level>
</Parameter>

```

The radio allows to chose between 3 radio bands

```

<Parameter>
  <id>P002</id>
  <domain>radio</domain>
  <name>band</name>
  <unit>MHz</unit>
  <default>868</default>
  <level>
    <selection>315</selection>
    <selection>433</selection>
    <selection>868</selection>
  </level>
</Parameter>

```

...and between 3 send powers measured In dBm

```

<Parameter>
  <id>P003 </id>
  <domain>radio</domain>
  <name>TX power</name>
  <unit>dBm</unit>
  <default>+10</default>
  <level>
    <selection>+10</selection>
    <selection>0</selection>
    <selection>-6</selection>
  </level>
</Parameter>
</Component>

```

This example is correct following the XSD description found in the Appendix of this document. The example shows that the description of diverse properties is possible in a readable and understandable form. It also demonstrates that hardware and software components can be modelled in the same format. Finally the transmit power property shows how properties can be computed using parameters and the given formula.

Indeed, writing such descriptions by hand is a tire-some task. It has been a motivation to develop support tools that help to create and maintain the databases. The tools are presented in the following section.

5 Tool support for the WSAE engineering

This section presents a set of tools that work on the data structures presented in the previous section. The tools support three operations as they are part of the general design flow presented in Section 2. The operations are the definition of the application requirements, composition of applications and evaluation of properties for the composed software. Additionally a tool to create and maintain the databases is presented.

It is the goal for all presented tools to find acceptance in the development community and within the WSAE4CIP consortium by openness, usability and system-independence. The XML structure presented in the previous section is a clear step towards such an openness and system-independence, and we are looking to keep up with the tools.

Figure 27 shows the current tool chain, consisting of three main operations: setting up the databases as task that has to be done before the actual WSAE engineering starts, and selection of the requirements and the actual composition as part of the application engineering process. The three operations are discussed in detail in the following subsections.

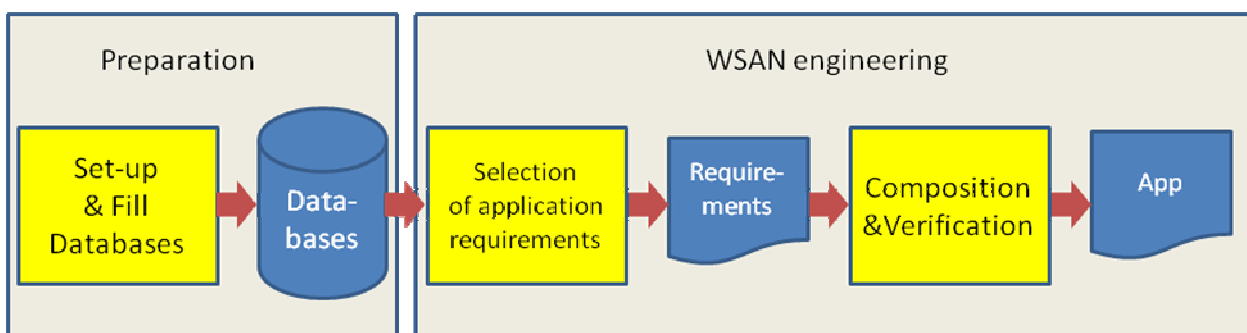


Figure 27: Chain of tool-supported WSAE engineering. Yellow boxes name the tools to generate the blue documents.

5.1 Set up of databases

One major problem we found for most model-driven analysis approaches is the absence of tools to set up and extend the fundamental databases. As introduced earlier, in our decisions process we need three databases:

- Requirement database which contains all potential requirements, constraints and environmental relations
- Component database containing all modules, interfaces with their properties, parameters and static heuristics
- Reasoning database as basis of factual reasoning from requirements and test results to an extended input to the selection process.

While the requirement database to some extent can be set up by us, i.e. the developers of the tool chain, describing all available components and their behaviour is not feasible for a small group of experts. First reason is the pure number of potential components that have to be categorized, and second without in-depth knowledge of the specific modules a reasonable model can hardly be realised. Also the reasoning database will only become valuable when different developers have entered various conclusions of their own experiences. Otherwise, if the database is only populated with experiences of few people, the reasoning process will be much likely biased and less objective.

Compared to other modelling languages, the XML scheme presented in Section 4 is already not too extensive, and the XML syntax is supported by many editors. Anyway we believe it is naive to assume that external developers would start to learn the semantics to add some modules. This is why we spent efforts to develop a GUI that allows to fill out the databases in an application frontend. The basic concept of the GUI was already implemented as part of the UbiSec&Sens project. In this work we mainly had adapt the new syntax and semantics of the new knowledge database concept.

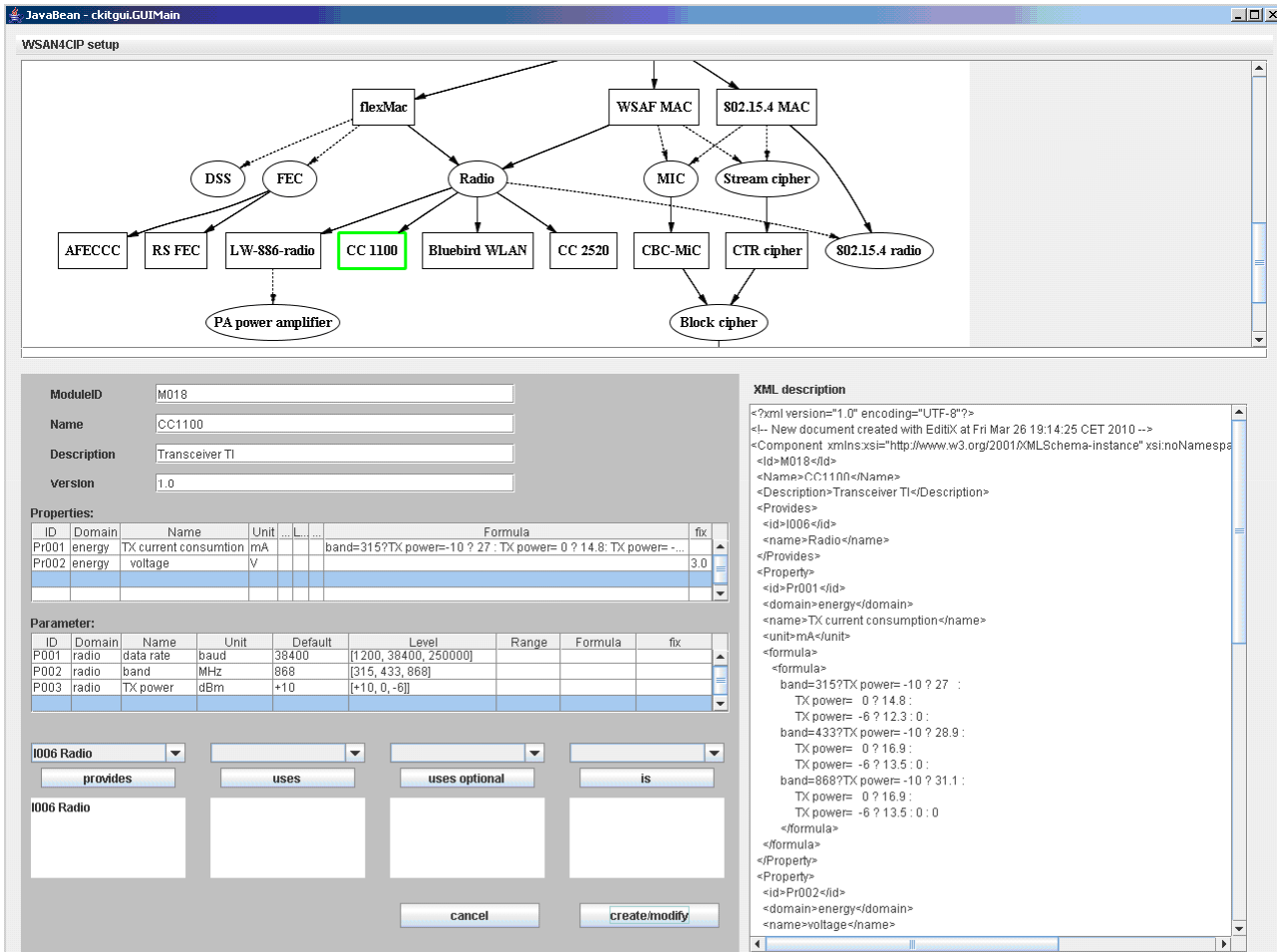


Figure 28: GUI of the components database editor

Figure 28 shows the GUI to edit the components database. On the top a user can see the components and their relations to other modules and interfaces. The modules can be selected to edit their properties in the bottom windows. The corresponding XML file will be generated and stored in the database so that it is available for the future selection processes.

Similar front ends have been designed for the requirements database and the reasoning database. Our evaluations show that they help us creating and maintaining database entries that are syntactically and semantically correct. Still there some weaknesses that could be fixed in future. For example entering long formulas for property computation without further assistance can be frustrating and error-prone. Also to improve the accessibility in future domain-specific templates could be implemented.

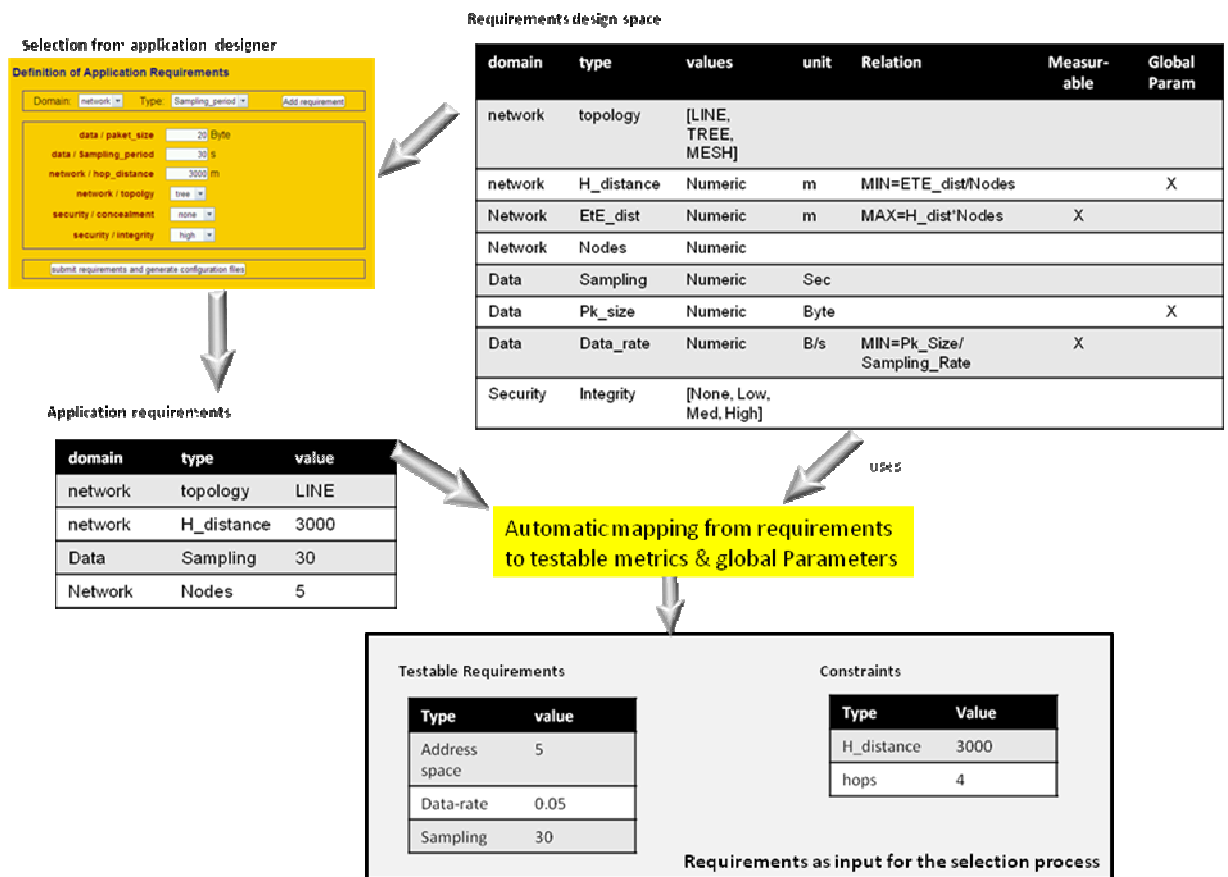
5.2 Selection of Application Requirements

The general idea of the requirements definition has been given in Section 2.3.2. An application designer can choose to define the application requirements and parameters using the requirements database. The requirements database contains all available requirements and their domain space. The notion is that requirements that are not part of the requirements database cannot be used for the application definition because they would not be supported in the further development flow anyway.

The setup of requirements is part of the database setup tools presented in the previous subsection. In this subsection we show the selection of the application requirements. Thus it is the second yellow box of Figure 27. The selection process is divided in two sub-operations:

1. The application developer chooses a set of application requirements
2. Mapping of the chosen requirements to testable metrics and global parameters

Figure 29 illustrates the process.



Datastructures - Requirements

Figure 29: Process and data structures of the requirement definition: Application designer chooses basic requirements in a form, automatic mapping creates tables as input to further processing.

The current application selection interface is web based, i.e. a designer can define the WSAN application in the favourite browser. An exemplary application selection in the user interface is shown as Figure 30.

Definition of Application Requirements

Domain: Type:

data / paket_size Byte

data / Sampling_period s

network / hop_distance m

network / topolgy

security / concealment

security / integrity

Figure 30: User interface for the selection of the application requirements

On the top, the application designer can chose the domain of the parameters, the select the parameter and add it to the list of requirements. The parameterisation will be done in the input field of the list. Eventually the dialog generates the Application description (see section 2.3.2.1).

The second step of the application definition process, the requirements given by the designer will be translated to the two requirement input tables - needed for the further processing. For generation of those tables the additional fields of the requirements database *measurable* and *globalparam* are used. Additionally each requirement that can be resolved (by the *relation* property) will be added to the active requirement tables. For example the required data rate can be derived from the packet size and the sampling rate. The sampling rate can be computed from the sampling period. As result of this multi-pass process the number of requirements and constraints (the global parameters) will increase significantly. It also means that the probability rises that answers will be found, because the problem statement eventually is given from many different perspectives.

This application requirements statement is the input for the selection and evaluation process.

5.3 Selection and Evaluation of a Suitable Module Composition

The task of the selection and evaluation tool is to compose and to configure available software and hardware modules, checks their functional compatibility and verifies compliance to the overall system requirements. The general problem statement is similar to the configKIT approach presented in UbiSec&Sens and in [34].

5.3.1 Differences to the Selection in UbiSec&Sens

While the basic selection algorithm is similar to the one presented in UbiSec&Sens, several modifications are necessary to respect new data structures and their new semantics. The most important changes are:

- functional classification is not only provided by abstract classes anymore
 - instead we use the interfaces of the modules for a functional description
 - background: the database in USS showed that about 90% of the modules had one interface that corresponded to the optional “is” description. So an SMAC module `_is_` a MAC and it provides a MAC interface. The ECDSA module `_is_` a digital signature, and it provides an interface for the computation of digital signatures. Shifting
- support of optional used interfaces
- parameterised interfaces
 - a component that uses an interface can define the conditions of the usage. If the providing module cannot comply with that parameters the modules will not be connected. That feature is new.
- different property representation
 - In the old structure we had specific properties hard-coded in the XML files. For example security-related properties had an individual data structure. Now all properties are part of the `paramType` structure. It is much more flexible and allows general treatment of all properties.
- No dedicated hardware modules
 - Hardware and software are part of the same component structure. A manual hardware selection corresponds to a pre-selection of some components. It improves the support of hardware accelerators within the selection process.

5.3.2 Selection Process

The requirements are the input for the selection process that computes a selection of modules that work together on the specified hardware and satisfy the application requirements.

5.3.2.1 Formal Problem Description

Considered we have the two descriptions:

- Available Modules as they are described in the Component Database:
 - Set of Interfaces I
 - Set of Modules M ($M=V\cup R$, $V\cap R=\emptyset$)
 - Set of (Virtual) Functional Groups V : $V\subseteq M$
 - Set of (Real) Implementations R : $R\subseteq M$
 - $\forall x\in R$: has a set of interfaces it provides (P_x) and uses (U_x)
 - $P_x\subseteq I$; $U_x\subseteq I$; both can be empty
 - $\forall x\in R$: has a set of Functional Groups F_x it belongs to
 - $F_x\subseteq V$; can be empty
- Application Description:
 - Pointer on either functional groups or implementing modules = Subset A of M : $A\subseteq M$

We are looking for all sets of modules C (configurations) where:

- $A\subseteq C\subseteq M$
- $\forall x\in(C\cap V): \exists y\in(C\cap F_y)$

Meaning that for every virtual module in the set there exists either a refinement (another functional group) or an implementation that implements the virtual module

- $\forall x \in (C \cap R): \forall i \in U_x: \exists y \in (C \cap R)$ so that
- Meaning that for every interface used by an implementation has to exist a module that provides the interface

While respecting additional requirements:

- All modules of a configuration have to be compatible to the selected hardware
- Measurable requirements have to be met

5.3.2.2 Selection Algorithm

The straightforward way of finding all configurations C satisfying the requirements is to use a backtracking algorithm. It starts with the application description A ($A \square M$) and traverses through the graph until either the requirements are fulfilled or no modules can be found that match a dependency.

STEP 1: (initialisation)

- Mark all modules that are explicitly required
- Remove modules from the selection process that are clearly not an option

STEP 2: (follow the requirements)

- if there are no open requirements (open interfaces) go to STEP 3
- mark all modules that are required by the currently marked modules.
 - mark all interfaces that are required by the currently selected modules
 - if the already selected modules can provide the interfaces go to STEP 3
 - look for modules in the database that can provide the required interfaces
 - in case there is more than one combination, successively try each (branch, mark, and go to STEP 3)
 - in case no satisfying combination could be found go back to previous branch (backtrack)

STEP 3: (follow the refinements)

- if there are no virtual modules without implementation go to STEP 4
- for each virtual module without implementation
 - look if a new module (added in STEP 2) provides the implementation
 - look for modules in the database that can provide the required implementation
 - in case there is more than one combination, successively try each (branch, mark, and go to STEP 4)
 - in case no satisfying combination could be found go back to previous branch (backtrack)

STEP 4: (evaluation)

- if there are no open requirements and no open virtual modules:
 - check additional requirements
 - if current configuration satisfies all needs store it
- go back to previous branch (backtrack)

At the end of this algorithm we get a list of suitable configurations, that all run on the selected node and that all satisfy the user's needs. Currently the selection algorithm chooses the smallest (regarding memory) as preferred configuration and lists the others in a separate window.

5.3.2.3 Optimisations

The presented algorithm in fact works while the run-time with a large number of modules can be unsatisfying. The major reason is that the arbitrary composition process will always be a non-polynomial (NP) prob-

lem. It can be simply proofed by the fact that if a module needs n other modules, and for each other module we have x alternatives with same parameters, so that no module can be eliminated, then we have x^n alternatives with equal quality that all have to be evaluated. Therefore it is the challenge for the selection algorithm to boil down the practical complexity by eliminating non-suitable design options early in the selection phase.

We realised it with the concept of disabling the visibility of components for the selection algorithm in case a module is no proper design decision at this point. Implemented reasons for the disabling modules from the design space are

If after a module has been added and the following recursion returns without having found a single configuration, the corresponding module can be blocked for all siblings and their sub-trees. The approach works well considered the final configuration is a tree. Then we know a component represents the own module plus all needed dependencies. The sub tree below the component would be considered as one static leaf. But many configurations contain reconvergences, i.e. modules are used by more than one other component.

Figure 31 shows such an example: the app uses two interfaces. Interface 1 is provided by component a, b, and c, while Interface 2 is provided by component d and e. Module c and e both use the interface 3, which in this case is only provided by module f.

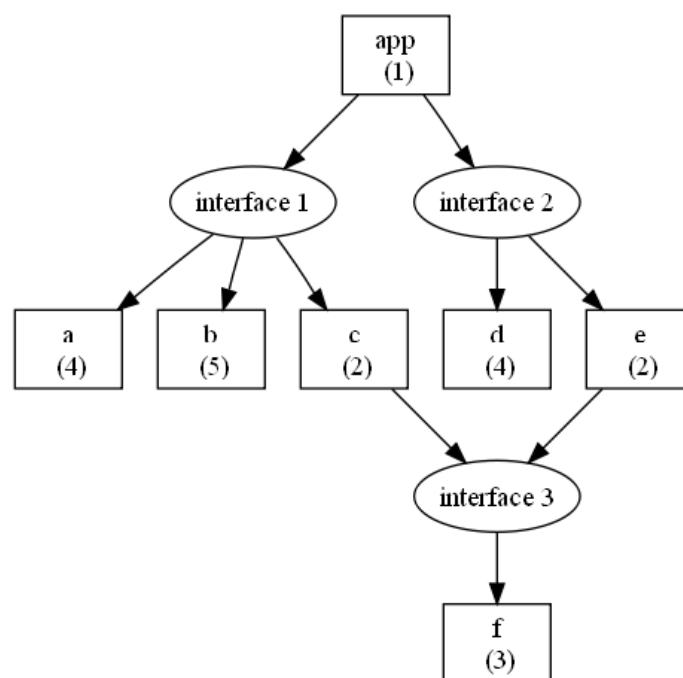


Figure 31: Example for re-convergences in the configuration graph. A combination of modules c, e and f is the smallest, considering app needs interface 1 and 2. The numbers in the brackets represent size.

Consider the numbers in the brackets represent sizes and we are looking for an app with the smallest size, i.e. the smallest sum of the sizes of the used modules.

A full search would result in the 6 possible configurations with the following resulting sizes:

- app, a, d = 9
- app, a, e, f = 10
- app, b, d = 10
- app, d, e, f = 11
- app, c, d, f = 10
- app, c, e, f = 8

Apparently the last configuration containing modules c, e and f (while c and e share f) would be the best one.

The naïve optimisation approach to disable modules whenever they do not show any benefits, while providing the same interface as another module would lead to the following sequence.

- app, a, d = 9
- app, a, e, f = 10 → disable e because it's worse than d
- app, b, d = 10 → disable b because it's worse than a
- app, c, d, f = 10 → disable c because it's worse than a

The resulting configuration containing modules a and d is the preferred one in this case, but the best solution (c,e,f) was ignored.

To solve the problem we added the following refinement to the optimisation of the selection algorithm:

The idea is that the decision for the selection of a component depends not only on the interfaces it provides but also on already selected components that already provide interfaces the surveyed component uses. For example, component

At an initialisation each module builds a list of all interfaces that can be used in the tree below the corresponding module.

$I[\text{app}] = \{1,2,3\}$

$I[\text{a}] = \{\}$

$I[\text{b}] = \{\}$

$I[\text{c}] = \{3\}$

$I[\text{d}] = \{\}$

$I[\text{e}] = \{3\}$

$I[\text{f}] = \{\}$

Whenever a decision has been made, the state of the required inputs will be stored and the decision will be ignored in case one of the dependent interfaces has a different state.

Following this logic we get the following decision flow:

- app, a, d = 9
- app, a, e, f = 10 → disable e, under condition that interface 3 is disabled
- app, b, d = 10 → disable b
- app, c, d, f = 10 → disable c, under condition that interface 3 is disabled
but with enabled c and f, interface 3 is enabled and that enables module e, so that combination c,f,e is an option
- app, c, e, f = 8

Advantage of the proposed optimisation is that it still finds the best solution. However the effectivity of the optimisation can suffer under that guarantee. Currently we are not aware of any other optimisation approach that can further improve the performance and still always finds the best solution. Fuzzy or genetic algorithms, simulated annealing, and neural networks have been discussed, but are not yet part of the development.

6 Examples

6.1 Radio properties

The approach of simplified heuristics is particularly challenged when it comes to properties and metrics that usually cannot be modelled in a small description. For example the prediction of awareness of specific radio properties and error codes in specific environments is a scientific discipline on its own.

The effectiveness of an error correction approach depends on the actual bit error rate and its burstiness. Even though interferences and error bursts are a huge problem in reality that property is even ignored in most sophisticated simulations. But already the prediction of an average BER cannot be considered as straightforward. In the following example we want to illustrate how the meta information embedded in the module description together with the environment parameters can help to select the right radio settings and to save energy. All described properties and formulas are part of the current databases, and are supported by the selection tool.

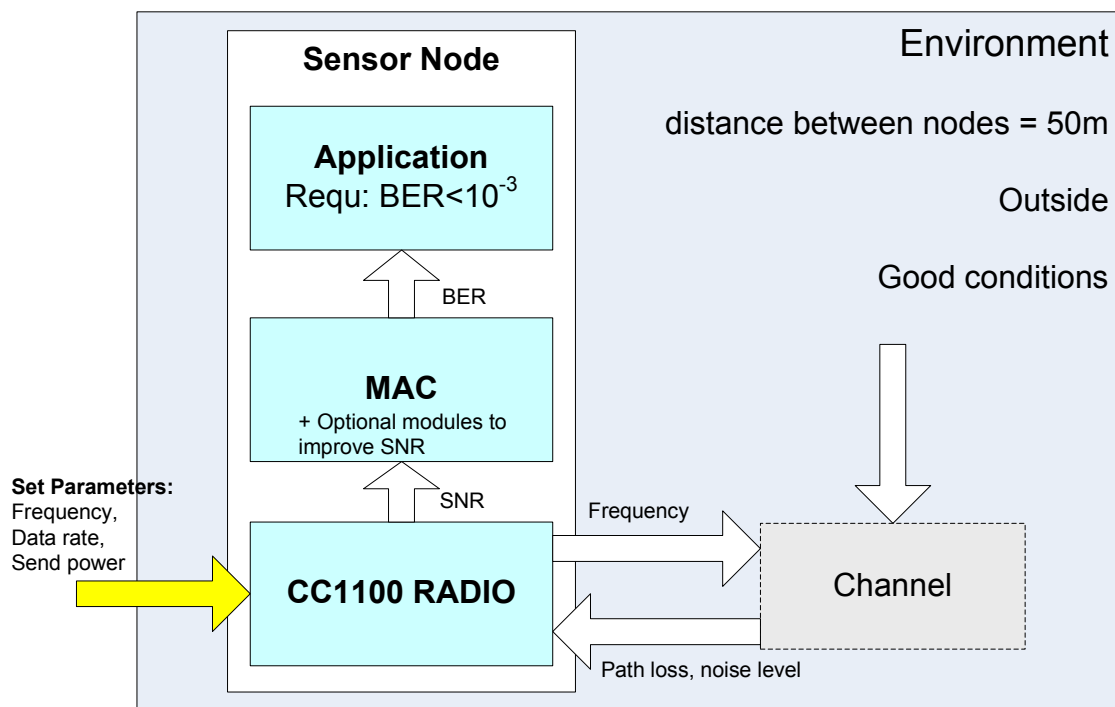


Figure 32: Model for the assessment of radio properties: sensor node with three components communicates of a channel defined by environmental parameters. Arrows show the information flow during the assessment process.

Considered we have the scenario as shown in Figure 32: In an environment (outside, good radio conditions) the nodes have a maximum distance of 50m. The sensor node has three components: the radio, a MAC layer and an application that directly uses the MAC. The requirement of the application is that the BER has to be smaller than 10^{-3} .

The question is how to determine whether a configuration of a radio and a MAC satisfies the given requirement in the environment.

Our model respects

- the send power of the sender (in dBm)
- the sensitivity of the receiver (in dB)
- the frequency used for the communication

- the free space path loss
- a background noise level
- error correction inside the MAC

The first three items are properties of the Radio.

In case of the TI CC1100 radio module [33] a developer can chose between

- the frequency band (315, 433, or 868MHz)
- the transmit power (-6, 0, 10 dBm)
- the data rate (1.2, 38.4, 250, 500kB/s) that affects sensitivity and receiving power consumption

We want to assume we decide for the 868 MHz band and the 38.4kB/s data rate, motivated by requirements not listed above. It sets the receiving sensitivity to -103dBm (computed as internal property of the CC1100 component, see Appendix A for the full specification).

The sending power, a free parameter, will be set to 0 dBm for the first iteration of the evaluation process.

With a radio frequency set by the radio module, the environment can compute the free space path loss. The simplified formula for the attenuation is

$$A = x \cdot \log_{10} \left(4\pi \frac{d}{\lambda} \right) = x \cdot \log_{10} \left(4\pi \frac{df}{c} \right)$$

While A is the attenuation in db, d is the distance in m, f is the used radio frequency (1/s), C is the speed of light (m/s), and x is a factor describing the extend of the attenuation. In vacuum and air it is usually set to 20. Close to the ground it is rather 40. We derive the factor from the environmental property (outside, good) and set it to 20. All factors (but f) are part of the requirements database. The formula itself is also described in the requirements database as relation.

For our example the attenuation is 65db.

In our case (sender sends with 0dBm and 65db attenuation) the signal level on receiver side will be -65 dBm. With -103 dBm receiving sensitivity, the signal level is +38dB.

But we still have the fifth parameter in our model: the environmental noise level. If the radio noise on receiver side is high, it will degrade the quality of the signal. The parameter is set by the environment and is measured in dBm. In our case (good, outside) it is assumed as -90 dBm. Then we get a signal-noise-ratio of -65 - (-90) = +25 dB

Since it is lower than the level without computed noise (the +38 dB) we have to take the +25 dB as SNR ratio. In case the environmental noise is lower than the sensitivity of the receiver, it would not be 'heard' by the receiver anyway and we have to take the former value. That is why the SNR is computed as

$$\text{SNR} = P_s - A - \text{MIN}(P_r, P_n)$$

While P_s is the sending power (in dBm), A is the attenuation, P_r is the receiving sensitivity, and P_n is the noise level.

The SNR is computed inside the radio module, using the environmental parameter A and pn. The model inside the radio is necessary because the SNR also depends on the applied modulation and other radio-specific properties. That is why SNR is often substituted by the term Eb/No which stands for Energy-per-bit per Noise-power-per-Hz-bandwidth. It can be modelled in the environment but in our example we simply assume to use the straightforward SNR.

The SNR is property of the radio and forwarded to the MAC module. The MAC can optionally enable some error correction to increase the SNR, before it computes a BER for the given SNR ratio.

A simplified model computes the Bit Error rate with the following formula.

$$\text{BER} = 0.5 \cdot \text{erfc}_{db}(\text{SNR}) = \text{erfc}(10^{(\text{SNR}/10)})$$

While $erfc_{db}(x)$ is the complementary error function based on the cumulative Gaussian distribution ($erfc_{db}(-\infty) = erfc(0)=1$; $erfc_{db}(-10) = erfc(0.1)=0.9$; $erfc_{db}(0) = erfc(1)=0.14$; $erfc_{db}(2) = erfc(1.5)=0.025$; $erfc_{db}(4) = erfc(2.5)=0.0004$; $erfc_{db}(6) = erfc(4)=10^{-8}$). The formula is a default math function of the tool.

The error function indicates that with an SNR of 6 and above the BER is de-facto 0. In our example with an SNR of +25 it is definitely the case.

The BER=0 would be forwarded to the app and there the application requirement is fulfilled. The conclusion of this computation is that the evaluated configuration with 0dBm sending power is a suitable configuration.

The model also allows to compute the power consumption of the configuration. The radio with the selected parameters needs 16.9 mA for sending and 15 mA for receiving. With a comfortable +25 dB receiving power it is one parameter we can ‘jeopardize’ for other properties.

For example sending with -6 dBm would reduce the sending consumption to 13.5 mA. That decision could be done either by brute force trying (try all parameters and see what we get) or by the reasoning table as described in Section 2.3.4.2

Performing all the steps above would result in a SNR of +19 dB and still a 0 BER, while reducing the sending power by more than 10 percent.

If we had a less cosy environment – assuming $x=30$ in the attenuation formula – the SNR would be -4 dB, resulting in a high error rate. Then the reasoning table would recommend to enable error correction in the MAC.

Critical comments to this example:

In the current model the channel is assumed to be part of the environment. For the simplified model we apply above it does work, while for more complicated channel models that approach is not recommended. Here a dedicated channel component, used by the radio module would be the preferred solution. An alternative solution for that issue would be the supplement of environmental modules.

The channel-assessment rules shown above are extremely simplified. For sure they cannot be applied for the development of new error codes or with high-end modulations. But usually a developer of WSNs is not interested in the development of new communication codes, but rather the major interest is whether or not a specific configuration can work for a given environment. Anyway the openness and flexibility of the proposed configuration scheme allows to extend the communication model to any level of detail.

6.2 WSAN4CIP FWA demonstrator

The FWA demonstrator is going to be one of the first real applications that will be designed with the proposed design flow. The general architecture of the intended system has been presented in Deliverable D1.2 and is shown as Figure 33.

This architecture is not output of any automated tool but result of human engineering. The question is, would an automated tool chain be able to ‘invent’ such an architecture? Most likely the answer is No, unless indeed the databases contain exactly the information to push the design tools to that architecture. An additional problem is that several components of the intended system are still under development. For central units such as the Node Control, the system’s architecture defines part of the behaviour and interfaces. They cannot be developed without knowledge of the architecture. The proposed design flow however needs exactly the interfaces and the behaviour of the components to develop the architecture. That imposes a circular problem that cannot be solved in the current tool chain. It is still necessary that human developers do design parts of the general architecture and define new components.

Once all components are modelled and part of the databases (entered by the set-up-tools described in section 5.1), we can enter the requirements as they were defined in Deliverable D1.1 [18] with the requirement selection tool described in section 0. Finishing the section process, the intended architecture has to be part of

Once a suitable configurations and parameters are found, we can evaluate properties of the configuration in dynamic tests and simulations as they are presented in Section 2. If the properties are satisfying the system can be deployed in the real environment. Following this design-methodology we expect to reduce the number of design errors and improve the chances for a first-time-right deployment of the system, because models and properties can be tested before the actual implementation, and the actual behaviour can be simulated before deployment.

Up to this point in the project we have modelled basic bodies for most of the components we are going to need in the FWA demonstrator. The corresponding system is shown as Figure 34.

One result of the current state of modelling the demonstrator is that beside the network interface also the interface to the flash memory is used by several components. As a result the resource has to be planned more carefully. As first reaction we added the requirement that the amount of flash used by the components (a simple additive metric) has to be not more than the amount of flash provided by the system. Each module using the flash is supposed to have a property indicating the amount of flash, and the hardware module will determine the available amount. Now that the property is stored in the active knowledge database it will be checked for each system that will be designed using this knowledge base.

In further development work we expect find more non-trivial properties whose management will be supported by the presented design flow.

7 Conclusion

In this deliverable we tackled a critical point in developing software and systems for wireless sensor and actuator networks in the context of critical infrastructure protection. We present a general design flow to help engineering WSAAN applications. The design flow is a general requirement-driven design and test loop which consists of three integral operations: a) top-down mapping of requirements to a reduced design space of design alternatives, b) systematic composition and implementation, and c) testing and verification the system under development.

Each operation is supported by a central knowledge base which systematically contains all information needed for the engineering process. The structure, motivated in Section 2 is finally extended to an open system-independent description language for WSAAN systems. It allows to describe requirements, environment, components and the whole system.

To support the usability and acceptance of the knowledge base we presented several tools to manage the content of databases. Further an open objective scheme to describe application requirements is given. The presented application requirement description scheme breaks down fuzzy requirements given by the system developers to precise measurable system metrics. They are the formal input needed for the actual application composition process. Furthermore the presented requirement definition can be applied as general language to characterize WSAAN application – even outside the presented tool-chain.

The proposed tool-supported composition process additionally relies on a component meta-model that combines properties from software engineering with domain-specific behaviour prediction. The presented model-driven scheme is able to predict the properties of the overall system based on static properties of components in context of the given environment. For more complex properties especially in the area of dynamic network behaviours we presented a simulation approach that combines network simulations with actual properties of the operating system and the software stack.

The examples in Section 6 showed the general practicability of the proposed approach. The radio property example includes a static channel assessment model to predict properties of a wireless channel. It highlights the possibilities to model even complex systems building on few model descriptions. The second example is related to one of the demonstrators of the WSAAN4CIP project. Even though it is work in progress it already shows that the presented tool-supported design flow can benefit the development process. The real value however will be demonstrated in the other technical work packages 2 to 5 of the WSAAN4CIP project where we expect the proposed design flow to be applied.

References

- [1] S. Moschoyiannis and M. Shields. A set-theoretic framework for component composition. In *Fundamenta Informaticae*, pages 59, 373-396, Amsterdam, The Netherlands, 2004. IOS Press
- [2] Thomas Genßler and Christian Zeidler. Rule-driven component composition for embedded systems. In *Intl. Conf. on Software Engineering (ICSE): Workshop on Component-Based Software Engineering*, Toronto, Canada, 2001
- [3] Kung-Kiu Lau. Component certification and system prediction: Is there a role for formality. In *Proceedings of the Fourth ICSE Workshop on Component-based Software Engineering*, Toronto, Canada, 2001, 80-83
- [4] P. Seymer, A. Stavrou, D. Wijesekera and S. Jajodia. Security Policy Cognizant Module Composition. Technical Report, Department of Computer Science, George Mason University, Fairfax, 2010
- [5] J. Ahn, S. Hong, and J. Heidemann. An adaptive FEC code control algorithm for mobile wireless sensor networks. In *Journal of Communications and Networks vol 7*, pages , 489, 2005
- [6] K. Römer and F. Mattern. The Design Space of Wireless Sensor. In *Networks IEEE Wireless Communications 11*, pages 54-61, 2004
- [7] N. Pantazis and D. Vergados. A Survey on Power Control Issues in Wireless Sensor Networks. IN *IEEE Communications Surveys & Tutorials, volume 9, n.4*, pages 86-107, 4th Quarter 2007
- [8] World Wide Web Consortium. XML - Extensible Markup Language. At <http://www.w3.org/XML>, March 2010
- [9] Chunying Zhao and Kang Zhang. Transformational Approaches to Model Driven Architecture - A Review. In *31st IEEE Software Engineering Workshop, (SEW 2007)*, pages 67-74, Columbia, Maryland, USA, 6-8 March 2007
- [10] Yashwant Singh and Manu Sood. Model Driven Architecture: A Perspective. In *IEEE International Advance Computing Conference (IACC 2009)*, pages 1644-1652, Patiala, India, 6-7 March 2009
- [11] Object Management Group. OMG Model Driven Architecture. At <http://www.omg.org/mda/>, Massachusetts, USA, 12. August 2009
- [12] Eclipse Foundation, Inc. Eclipse Model Development Tools (MDT). At <http://www.eclipse.org/modeling/mdt/?project=uml2>, Ottawa, Ontario, Canada, 2010
- [13] Eclipse Foundation, Inc. Eclipse Modeling Framework Project (EMF). At <http://www.eclipse.org/modeling/emf/>, Ottawa, Ontario, Canada, 2010
- [14] Nils Hoeller, Christoph Reinke, Jana Neumann, Sven Groppe, Christian Werner and Volker Linne-mann. XML Data Management and XPath Evaluation in Wireless Sensor Networks. In *Proceedings of the 7th International Conference on Advances in Mobile Computing & Multimedia (MoMM2009)*, Kuala Lumpur, Malaysia, 2009
- [15] Karl Aberer, Manfred Hauswirth and Ali Salehi. Zero-Programming Sensor Network Deployment, In *IEEE International Symposium on Applications and the Internet Workshops, (SAINT Workshops 2007)*, Hiroshima, Japan, 2007
- [16] Mohammad Al Saad, Elfriede Fehr, Nicolai Kamenzky and Jochen Schiller. Scatterclipse: A Model-Driven Tool-Chain for Developing, Testing, and Prototyping Wireless Sensor Networks. In *6th IEEE International Symposium on Parallel and Distributed Processing with Applications, (ISPA'2008)*, pages 871-885, Sydney, Australia, 2008
- [17] G. Karsai, J. Sztipanovits, A. Ledeczki and T. Bapty. Model-integrated development of embedded software. In *Proceedings of the IEEE*, pages 91, 145-164, January 2003
- [18] WSAN4CIP project, Deliverable D1.1- Definition of critical performance and reliability parameters through radio coverage, energy and CIP process time tomography. At http://www.wsan4cip.eu/fileadmin/public_documents/Deliverables/WSAN4CIP_D1.1_Def.of.parameters.pdf, 2009

- [19] WSAN4CIP project, Deliverable D1.3- Modular design and performance ranking of communication protocols. At http://www.wsan4cip.eu/fileadmin/public_documents/Deliverables/D1.3_Performance_of_protocols_WSAN4CIP-225186.pdf, 2010
- [20] UbiSec&Sens, Ubiquitous Sensing and Security in European Homeland. At <http://ist-ubiseconsens.org>, Heidelberg, Germany, 2009
- [21] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer and D. Culler. Tinyos: An operating system for sensor networks. In *Ambient Intelligence*, pages 115–148. 2005
- [22] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson and Richard Han. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. In *Mobile Networks and Applications vol. 10(4)*, pages 563–579, 2005
- [23] Chris Cleeland, Douglas C. Schmidt and Tim Harrison. External polymorphism - an object structural pattern for transparently extending c++ concrete data types, 1997
- [24] TinyOS Community. Tinyos community forum. At <http://www.tinyos.net/>, November 2009
- [25] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462, November 2004
- [26] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, 2003, ACM Press
- [27] Norman C. Hutchinson and Larry L. Peterson. The X-Kernel: An architecture for Implementing Network Protocols. In *IEEE Transactions on Software Engineering vol. 17(1)*, pages 64–76, Los Alamitos, California, USA, 1991.
- [28] Kevin Klues, Gregory Hackmann, Octav Chipara and Chenyang Lu. A Component-Based Architecture for Power-Efficient Media Access Control in Wireless Sensor Networks. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 59–72, New York, NY, USA, 2007. ACM
- [29] Philip Levis, Nelson Lee, Matt Welsh and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM
- [30] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne and T. Voigt. Cross-level sensor network simulation with cooja. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 641–648, November 2006
- [31] Adi Mallikarjuna V. Reddy, A.V.U. Phani Kumar, D. Janakiram and G. Ashok Kumar. Wireless sensor network operating systems: a survey. In *International Journal of Sensor Networks 2009 vol. 5 Nr. 4*, pages 236–255, Geneva, Switzerland, 2009, Inderscience Publishers
- [32] Christian Tschudin. Flexible protocol stacks. In *SIGCOMM '91: Proceedings of the conference on Communications architecture & protocols*, pages 197–205, Zürich, Switzerland, 1991. ACM
- [33] Texas Instruments. CC1100 Low-Power Sub- 1 GHz RF Transceiver. At <http://focus.ti.com/lit/ds/symlink/cc1100.pdf>, Dallas, Texas, 2006
- [34] S. Peter, K. Piotrowski, and P. Langendörfer. In-network-aggregation as case study for a support tool reducing the complexity of designing secure wireless sensor networks. In *Proceedings of the Third IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenSeApp'08)*, Montreal, Canada, October 2008

Annex A Scheme of WSAN4CIP Components and Requirements

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:annotation>
    <xs:documentation>This Schema defines the structure of WSAN4CIP Modules</xs:documentation>
  </xs:annotation>
  <xs:element name="Component">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="ComponentType">
          <xs:sequence/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="Interface">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="ComponentType">
          <xs:sequence/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="ComponentType">
    <xs:sequence>
      <xs:element name="Id" type="xs:string"/>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Description" type="xs:string" minOccurs="0"/>
      <xs:element name="Is" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="id" type="xs:string"/>
            <xs:element name="name" type="xs:string" minOccurs="0"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Provides" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="id" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Uses" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="id" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="parameter" minOccurs="0">
              <xs:complexType>
                <xs:complexContent>
                  <xs:extension base="ParamType">
                    <xs:sequence/>
                  </xs:extension>
                </xs:complexContent>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

```

```

        <xs:attribute name="optional" type="xs:boolean"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="Property" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:complexContent>
          <xs:extension base="ParamType">
            <xs:sequence/>
          </xs:extension>
        </xs:complexContent>
      </xs:complexType>
    </xs:element>
    <xs:element name="Parameter" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:complexContent>
          <xs:extension base="ParamType">
            <xs:sequence/>
          </xs:extension>
        </xs:complexContent>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="Version" type="xs:string"/>
</xs:complexType>
<xs:element name="Requirements">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Requirement" maxOccurs="unbounded">
        <xs:complexType>
          <xs:complexContent>
            <xs:extension base="ParamType">
              <xs:sequence>
                <xs:element name="relation" type="xs:string" minOccurs="0"/>
                <xs:element name="measurable" type="xs:boolean" minOccurs="0"/>
                <xs:element name="globalParam" type="xs:boolean" minOccurs="0"/>
              </xs:sequence>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="Version" type="xs:string"/>
  </xs:complexType>
</xs:element>
<xs:complexType name="ParamType">
  <xs:sequence>
    <xs:element name="id" type="xs:string"/>
    <xs:element name="domain" type="xs:string"/>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="unit" type="xs:string" minOccurs="0"/>
    <xs:element name="default" type="xs:string" minOccurs="0"/>
    <xs:choice>
      <xs:element name="level" type="ChoiceType"/>
      <xs:element name="range" type="ValueType"/>
      <xs:element name="formula" type="FormulaType"/>
      <xs:element name="fix" type="xs:string"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="ValueType">
  <xs:sequence>
    <xs:element name="min" type="xs:float" minOccurs="0"/>

```

```
        <xs:element name="max" type="xs:float" minOccurs="0"/>
        <xs:element name="resolution" type="xs:float" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="FormulaType">
    <xs:sequence>
        <xs:element name="formula" type="xs:string" minOccurs="0"/>
        <xs:element name="resolution" type="xs:float" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="ChoiceType">
    <xs:sequence>
        <xs:element name="selection" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
</xs:schema>
```