TECHNICAL UNIVERSITY OF CRETE
ELECTRONIC AND COMPUTER ENGINEERING DEPARTMENT
DIVISION OF COMPUTER SCIENCE

Model Driven Development in Sensor Networks

by

Rontidis Pavlos

A thesis submitted in partial fulfillment of
the requirements for the diploma degree of

ELECTRONIC AND COMPUTER ENGINEERING

June 2013

THESIS COMMITTEE

Assistant Professor Vasilis Samoladas, *Thesis Supervisor*
Assistant Professor Antonios Deligiannakis
Associate Professor Yannis Papaefstathiou

Abstract

Sensor networks is an emerging, active research area with numerous applications. Despite their popularity, there are many implementation challenges due to the fact that developing a sensors network application is a complex process that requires expertise in various fields. The special needs of each application domain, the distributed nature of such networks, the hardware restrictions and radio protocols are common factors that make the separation of concerns very difficult. The purpose of this diploma thesis is to enable model driven development in the domain of sensor networks by providing a high-level domain specific language and a programming tool capable of generating code. This development tool allows for quick system development which promotes the re-usability of software components and libraries while guaranteeing true functional and behavioral portability among different hardware platforms and vendors. Programming in a high-level, domain specific language enables the developers to focus on the application domain and its specific features while the low-level, technical aspects are handled by other specialists that implement the middleware or the hardware abstraction libraries.

Keywords: Model Driven Development, Model Driven Architecture, Domain Specific Language, Model Transformation, Code Generation, Wireless Sensor Networks, Eclipse Modeling Project

Thesis Supervisor: Vasilis Samoladas
Title: Assistant Professor

Acknowledgements

Several people have been of great a help to during my 5 year studies. Each person, in his\her unique way contributed to a well-rounded university life.

First of all, I want to express my deep gratitude to my family that enabled my studies with their consistent support. My father's optimism about my future, my mother's emotional support as well as the admiration of my little brother were invaluable.

Secondly, I want to thank my academic advisor, Vasili Samolada, for his guidance and support. His abilities and experience enabled me to overcome the most difficult obstacles I faced.

Last but not least, I want to thank my girlfriend and close friends that helped me achieve balance in my life. This balance was source of my tranquility that enabled me to deal with the majority of daily problems in a calm and successful manner.

Table of Contents

Chapter 1

Introduction

1.1 Wireless Sensor Networks

Wireless sensor networks (WSNs) have gained worldwide attention in recent years. These networks are comprised of small sensors, with limited processing and computing resources. These sensor nodes can sense, measure, and gather information from the environment and, based on some local decision process, they can transmit the sensed data to a sink node or a server.

The main unit of a WSN is a sensor node. Sensor nodes are low power devices equipped with one or more sensors, a processor, memory, a power supply, a radio, and an actuator. A variety of mechanical, thermal, biological, chemical, optical, and magnetic sensors may be attached to the sensor node to measure properties of the environment. Since the sensor nodes have limited memory and are typically deployed in difficult-to-access locations, a radio is implemented for wireless communication to transfer the data to a base station (e.g., a laptop, a personal handheld device, or an access point to a fixed infrastructure).

The autonomy of a sensor is an important issue. Battery is the main power source in a sensor node. In some case, additional power may be harvested power from the environment. For example, solar panels may be added to the node depending on the appropriateness of the environment where the sensor will be deployed. Depending on the application and the type of sensors used, actuators may be incorporated in the sensors.

The ideas inspired from such networks and the vast number of possible applications resulted in a creation of a big community that works on this field. However, several implementation challenges still exist and hinder the WSN application development. Several different specialists should cooperate

for the successful deployment of a WSN application. The application domain is the responsibility of experts such biologists, geologists, environmental engineers with no knowledge on WSN platforms. The other layers are in the context of software, hardware and telecommunications engineering.

1.2 An MDD Solution

This thesis follows the paradigm of model driven development and provides a code generation tool in order to develop a WSN application. Taking into consideration the aforementioned implementation difficulties and the fact that a team of various professionals may take part in building and deploying a WSN application, the motivation for this thesis was to completely separate the implementation concerns.

This work introduces a DSL -named SensL- that provides a new language that aids the programmer to develop an application without worrying about synchronization issues. Moreover, it provides a SensL to nesC transformation that is performed with the EMP tools. To achieve this, the SensL text is parsed into an AST, the AST is transformed into a PIM and the PIM is transformed into a nesC PSM. Then, the nesC PSM is used for generating code that can be compiled with an existing compiler. Finally, this approach requires a runtime component that schedules the event processing, regulates the rules' order of execution and manages the memory allocation.

To summarize, the developer writes a SensL program that is automatically transformed into a nesC model. The platform specific model is used to generate nesC code. The genrated nesC code in combination with the runtime component is compiled into an application that can be installed on motes.

1.3 Introduction to the SensL DSL

This thesis introduces a DSL that is used to write WSN applications. It allows the domain experts to describe a solution in a high level language

when compared to a language such as C. A SensL editor is provided with syntax-coloring and error detection features to make the development process easier.

SensL is adjacent to an object-oriented language like Java but it includes additional concepts such as module, ECA rule and frame. One can write a SensL program by hand and then invoke the tool to generate code for a specific platform. In the future, SensL may be used as an intermediate phase in the MDD process by providing a tool that generates SensL given a even higher level model which was created with a graphical modeling tool.

1.4 Thesis impact on the real world

The tools and results that were created in the context of thesis will contribute to the WSN-DPCM project. WSN-DPCM is a cooperation project of several technical universities and companies from Spain, Italy, Lithuania and Greece. The project is funded by the ARTEMIS Joint Undertaking (the European technology platform representing the field of advanced research and technology for embedded intelligence and systems), national authorities and European partner companies.

The objective of WSN-DPCM is to develop a full platform to address the main WSN challenges for smart environments that include the middleware for heterogeneous wireless technologies and an integrated engineering toolset for Development, Planning, Commissioning, and Maintenance activities for expert and non-expert users. To further increase the value for the field, most of the project development will be released under a suitable open source license for mutual benefit and to foster academic research and know how to transfer to industry.

Chapter 2

Related Work

2.1 Model Driven Development

The Model Driven Development is an software development methodology which focuses on creating and exploiting domain models. MDD gives architects the ability to define and communicate a solution while creating artifacts that become part of the overall solution. In the context of MDD, model is code i.e. the modeling languages take on the role of implementation languages, analogous to the way that third-generation programming languages displaced assembly languages. Therefore, MDDs full benefits can be attained when the automatic generation of complete programs from models is possible.The main motivation behind adopting such an approach is to improve productivity. Another motivation is the communication of a specific solution among big teams without ambiguities.

Some of the better known MDD initiatives are the Object Management Group (OMG) initiative Model-Driven Architecture (MDA) and the Eclipse ecosystem of programming and modelling tools. Standardization provides a significant impetus for further progress because it codifies best practices, enables and encourages reuse, and facilitates interworking between complementary tools. It also encourages specialization, which leads to more sophisticated and more potent tools. Still, with all the benefits of automation and standardization, model-driven methods are only as good as the models they help us construct.
In this thesis, the paradigm of Model Driven Architecture approach is the main guide on how to separate abstraction layers. There are four main principles that underlie the OMGs MDA approach:

- Models expressed in a well-defined notation are a cornerstone to system

understanding for enterprise-scale solutions.

- Building system can be organized around a set of models by imposing a series of transformation between models, organized into an architectural framework of layers and transformations.
- A formal underpinning for describing models in a set of metamodels facilitates meaningful integration and transformation among models, and is the basis for automation through tools.
- Acceptance and broad adoption of this model-based approach requires industry standards to provide openness to consumers and foster competition among vendors

An MDA overview is depicted in the figure **??**.

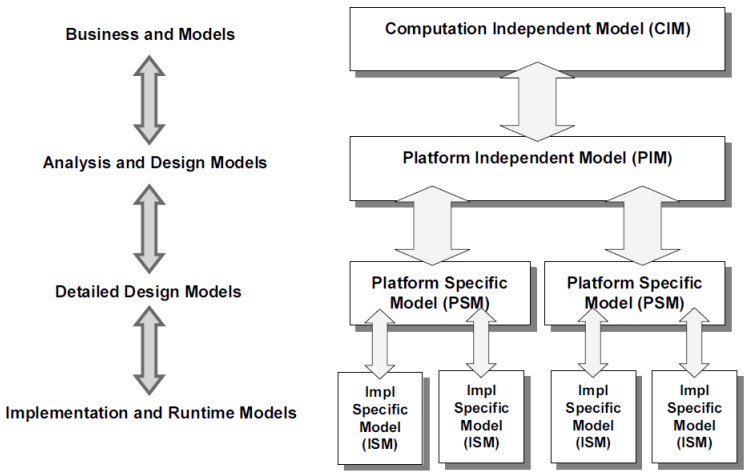


Figure 2.1: Overview of MDA

Specifically, in this thesis the application domain is modeled in the Platform Independent Model (PIM) which focuses only on the problem domain. Then, the PIM can be mapped to a Platform Specific Model (PSM) which contains details about the platform. The PIM to PSM transformation is executed my using a Model to Model transformation tool. The PSM probably requires a Model to Text transformation in order to produce code ready for compilation by utilizing preexisting compilers and tools.

2.2 Applying MDD to the WSN field

Several benefits emerge from the applying MDD to the WSNs field and they will be discussed in this subsection. Currently, the are many platforms (e.g. TinyOS, Contiki, COUGAR, SensOS etc.)each one having its own requirements, execution, programming environments and software tools that differ slightly or greatly. The platform is chosen according the special needs of each project.

The key thing to note is that the application domain is independent from the platform chosen by the developers. Thus, the application logic can be always described with the same PIM. This is not the case for the PSM because it contains details about each specific platform and a different PSM must be created for each target platform. Consequently, the PIM to PSM transformation also changes. The same applies to the code generation, it is dependent on the PIM and the programming language, thus different implementation of generators are needed. Thus, for the MDD approach to support a platform a PSM, a PIM to PSM transformation and a code generator are needed. However, all the aforementioned artifacts are implemented only once and are reusable.

The strong coupling between the application logic and the underlying sensor platform, along with the lack of a methodology to support the development lifecycle of WSN applications resulted in projects with platform dependent code that are hard to maintain, modify, and reuse. Although the initial overhead required by the MDD approach may be discouraging, the benefits from adopting it are clear.

After defining a high-level domain specific language (DSL) and a PSM and after implementing a code generator, the development of a WSN application speeds up significantly. Developers can focus on the problem at hand in a higher abstraction level without shifting their focus from the problem to the low-level platform details. This greatly reduces the complexity of devising a solution and also reduces the time spent on dealing with implementation difficulties. Also, the programmer's\team's effort is reduced in all the development phases which are now decoupled. Finally, the MDD ap-

proach enables low-cost prototyping, optimization and hardware-in-the-loop simulation.

The following figures depict examples of how was MDD applied to WSN applications.

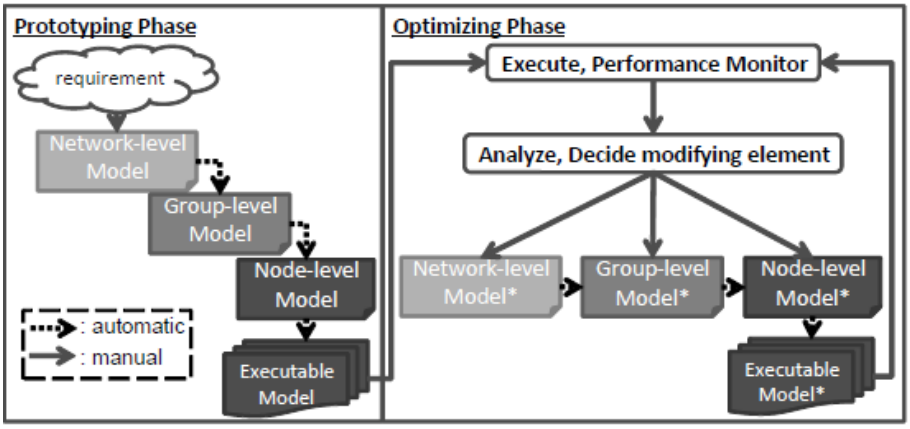


Figure 2.2: A proposed development process for prototyping, Ref:[3]

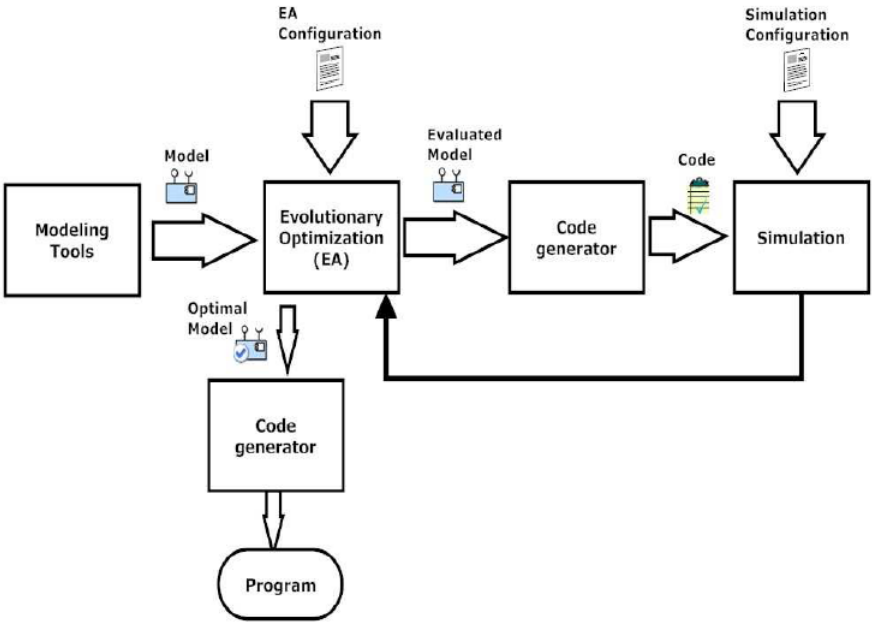


Figure 2.3: Another proposed development framework, Ref:[5]

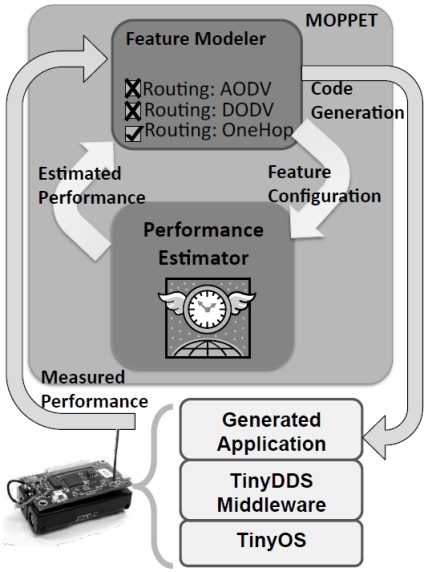


Figure 2.4: An Overview of the Moppet framework, Ref:[6]

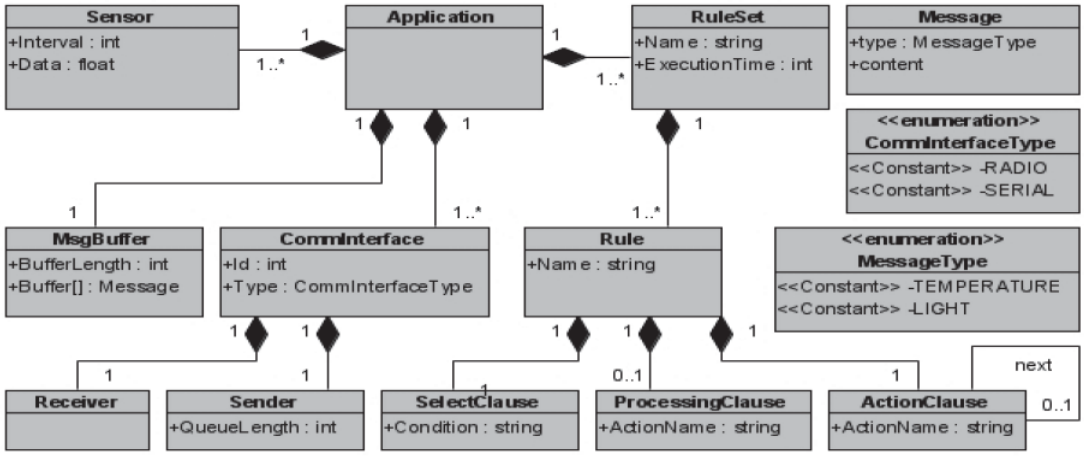


Figure 2.5: A proposed DSL meta-model for WSN applications, Ref:[4]

2.3 Eclipse Modeling Project

All the utilities and tools that were implemented as part of this work, including the meta-models and model transformations, have been developed using the MDE facilities provided by the Eclipse platform. The Eclipse Modeling Project (EMP) focuses on the evolution and promotion of model-based devel-

opment technologies within the Eclipse community by providing a unified set of modeling frameworks, tooling, and standards implementations. This free open-source environment offers one of the most widely used implementation of the OMG standard Meta-Object Facility (MOF), called Eclipse Modelling Framework (EMF). Although EMF currently supports only a subset of MOF, called EMOF (Essential MOF), it allows designers to create, manipulate and store both models and meta-models. This is the reason why many MDE-related initiatives are currently being developed around Eclipse and EMF.

2.3.1 Eclipse Modeling Framework

The core EMF framework includes a meta model (Ecore) for describing models and runtime support for the models including change notification, persistence support with default XMI serialization, and a very efficient reflective API for manipulating EMF objects generically. An overview and a more detailed version of the Ecore metamodel are depicted in the figures 3.1 and the 3.2.

In this thesis, both the PIMs and the PSMs conform to the EMF's Ecore metamodel for various reason. First of all, it is simple yet sufficient to define the PIM and the PSMs. Also, it is very useful because most of the MDD tools and all of the EMP tools support this metamodel by default and interoperate by using it.

2.3.2 Plug-in Development Environment

The Plug-in Development Environment (PDE) provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins, fragments, features, update sites and RCP products.PDE also provides comprehensive OSGi tooling, which makes it an ideal environment for component programming, not just Eclipse plug-in development.

The PDE subproject is broken down into three main components, Build, UI and API Tools. Each of these components operate like a project unto its own, with its own set of committers, bug categories and mailing lists.

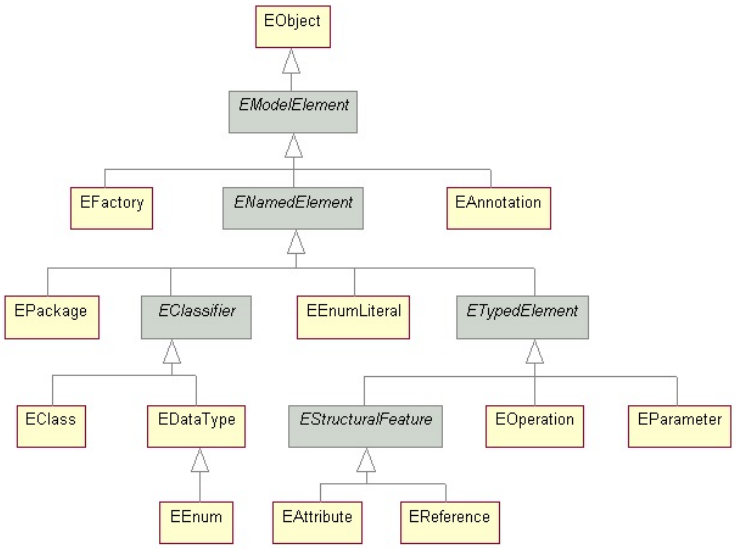


Figure 2.6: Overview of org.eclipse.emf.ecore

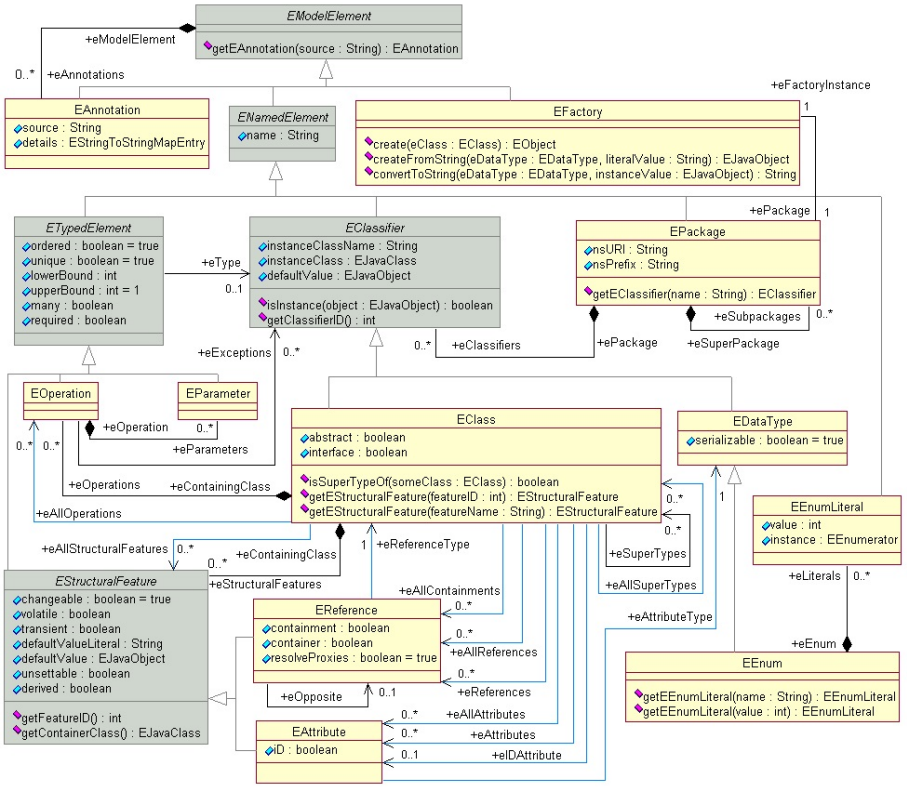


Figure 2.7: Detailed version of org.eclipse.emf.ecore

There are two additional components in PDE, Doc which handles the help documentation and Incubator which develops non-SDK features.

All the EMF Models that were created in this thesis are part of a PDE project. Additionally, for each model an editor can be generated by using the EMF Generator Model. The project is an Eclipse plug-in which can be installed in a standard Eclipse platform.

2.4 Xtext

Xtext is a framework for development of programming languages and domain specific languages. It covers all aspects of a complete language infrastructure, from parsers, over linker, compiler or interpreter to fully-blown top-notch Eclipse IDE integration.

Xtext provides the developer with a set of domain-specific languages and modern APIs to describe the different aspects of the created programming language. Based on that information it gives a full implementation of that language running on the JVM. The compiler components of the language are independent of Eclipse or OSGi and can be used in any Java environment. They include such things as the parser, the type-safe abstract syntax tree (AST), the serializer and code formatter, the scoping framework and the linking, compiler checks and static analysis aka validation and last but not least a code generator or interpreter. These runtime components integrate with and are based on the Eclipse Modeling Framework (EMF), which effectively allows the developer to use Xtext together with other EMF frameworks like for instance the Graphical Modeling Project GMF.

2.5 Model to Model Transformation

Model to model transformation is a crucial phase in the MDD process. There are many approaches available and in this thesis a hybrid approach is adopted that combines several MT approaches such as direct, operational, and template-based approach. The EMP provides the QVTo tool that is very

suitable for the implementation of this phase.

To begin with, each source and target (output) model must conform to a metamodel. This is not only good practice but it is also a requirement for using QVTo. In addiotion, the cardinality of input-output models in general may be 1-1, 1-N, N-1, N-N. All transformations presented in this thesis are 1-1. In the following figure a high level concept of the MT is depicted.

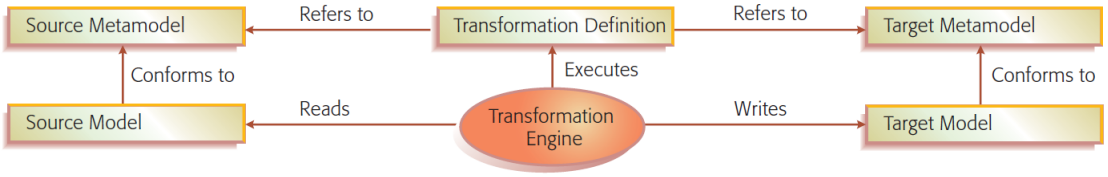


Figure 2.8: Basic concepts of Mode l Transformation [8]

The EMP provides tools which are suitable to perform the aforementioned types of model to model transformations. The **MMT** project hosts Model-to-Model Transformation languages and is a subproject of the top-level Eclipse Modeling Project. The transformations are executed by transformation engines that are plugged into the Eclipse Modeling infrastructure. Additional approaches and further details can be found in the references.

Specifically, the **ATL**(ATL Transformation Language) is a model transformation language and toolkit. In the field of Model-Driven Engineering (MDE), ATL provides ways to produce a set of target models from a set of source models. Another alternative is the **QVTo** which is a partial implementation of the Operational Mappings Language defined by the OMG standard specification (MOF) 2.0 Query/View/Transformation. In long term, it aims to provide a complete implementation of the operational part of the standard.

The QVTo was chosen to implement the M2M part of this thesis. QVT is a hybrid approach to Model Transformation with separate components that has 3 advantages:

- Standard: An OMG specification (currently v1.1) of this approach is available. This set of transformation languages is standardized and more mature than the other transformation alternatives.

- Support: It is part of the M2M project which a subproject of the Eclipse Modeling Project. Several parties (such as IBM, Unisys, France Telecom and university labs) have shown interest in this project so it unlikely to be dropped in the near future.

- Practical: It is implemented and integrated in Eclipse Modeling Tools, a portable and relatively mature tool. It can perform operations on Ecore models by default, it supports OCL operations and has a mature editor. Note that ATL is also integrated in EMT.

2.6 Model to Text Transformation

The final step of the MDD process is code generation and the combination of the produced source files with the runtime component. From the tools available in EMP, XPand was chosen for the code generation phase. It has many useful features such as Ecore support, simplicity of template definition and a good editor.

The Model to Text (M2T) project focuses on the generation of textual artifacts from models. Its purpose is threefold: First of all, provide implementations of industry standard and defacto Eclipse standard model-to-text engines. Secondly, provide exemplary development tools for these languages. Thirdly, provide common infrastructure for this languages.

There are many alternatives in M2T such as JET, Acceleo, XPand etc. For the purposes of this thesis, XPand was used. **Xpand** is a statically-typed template language featuring polymorphic template invocation, aspect oriented programming,functional extensions, a flexible type system abstraction, model validation and more. It comes with a good editor that offers useful features like syntax coloring, error highlighting, navigation, refactoring and code completion.

It is noteworthy that it supports Ecore model as input and the editor can auto-complete by using information from the model. XPand features are ideal for generating code from a model that conforms to a metamodel with Ecore as its meta-meta-model. It also allows to write Java code in order to

create utilities that are not available in the tool. The aforementioned reasons pushed towards choosing XPand for the model to text transformation.

Chapter 3

Modeling the problem domain

The separation of concerns during the development phases is very important due to the reasons explained in the previous chapter. The MDD approach requires a high-level DSL in order to decouple the application domain from the low-level software and hardware implementation. For this reason, the SensL DSL was created. The next sections explain the SensL terms and elaborate on the details.

3.1 SensL Terminology

ECA rules: An Event\Condition \Action rule is defined within a module and it consists of the following parts:

- **Event:** Events are objects defined by the programmer and indicate that an important -i.e. the functionality of a node is affected- event happened. It has a body inside which properties can be declared. An event triggers a rule invocation and can be emitted by rules or resources.
- **Condition:** A boolean expression that its value determines the execution of the ECA rule's action.
- **Action:** The action is the rules logic and it may:
  a. Update local properties
  b. Invoke methods
  c. Emit new event
  A scope for the action body must be defined. A rules action body has access to:
  1. Local variable or methods

2. Variables that belong to the rule's module.
3. Variables that belong to used modules.

However, there are more preconditions for a rules execution. As mentioned, the expression defined in the condition must be TRUE. Also, the module of the rule must be active, i.e. its mandatory frame must be enabled. See the Frame section for additional information.
Finally, if the condition part of an ECA rule is evaluated to TRUE the rule is enabled and its corresponding action part must be executed.
A valid rule structure is the following:

```
onEvent foo(args) {
when{ condition }
do{ action }
}
```

Table 3.1: ECA rule structure

Module: A module is the SensL component that defines overall function framework. A valid module structure is the following:

At any given time during execution a module has a state. The module states are:

- Enabled module iff modules condition is TRUE.
- Enabled rule iff enabled module and rule condition is TRUE.
- Enabled submodule iff parent module is enabled and submodule condition is TRUE.

The idea of enabled/disabled modules implicitly states that a specific modules functionality may change according to the state of the modules rules i.e. different combinations lead to the execution of different rules. The order of execution may also alternate the modules functionality.

Submodule: A module defined within another module. As implied by the definition, a submodule has the same structure and states with a module.

```
Module Foo {
property p1, , pN;
frame f;
uses module1, , moduleN;
uses resource1, , resourceN;
condition expression;

rule r1  action m1 ;
rule r2  action m2 ;
...
rule rN  action mN ;
method m1;
method m2;
...
method mN;
}
```

Table 3.2: Module structure

Module Hierarchy: Modules form a tree structure where the root is a module, the internal nodes are also modules and the leaves are rules. This tree can also be used to see illustrate when a submodule is enabled. The following figure illustrates a sample tree.

Resource: A resource is a top level component that represents a hardware (e.g. LED) or software (e.g. thread) resource. It may have properties, classes, modules, other resources and also it can emit events.

Node: A node is a top level component. It is a collection of modules and resources that defines a functionality. Enabling/Disabling a nodes module results in functionality change too. A node may represent the following: Mote, Base station, Server module.

Package: A package is a set of SensL components and is similar to a Java package. SensL packages are used in order to provide namespace separation to components. A collection of basic SensL components which comprise an overall node functionality can be included it in a package. Thus, packages are used to organize and separate the overall different operational modes of a node.

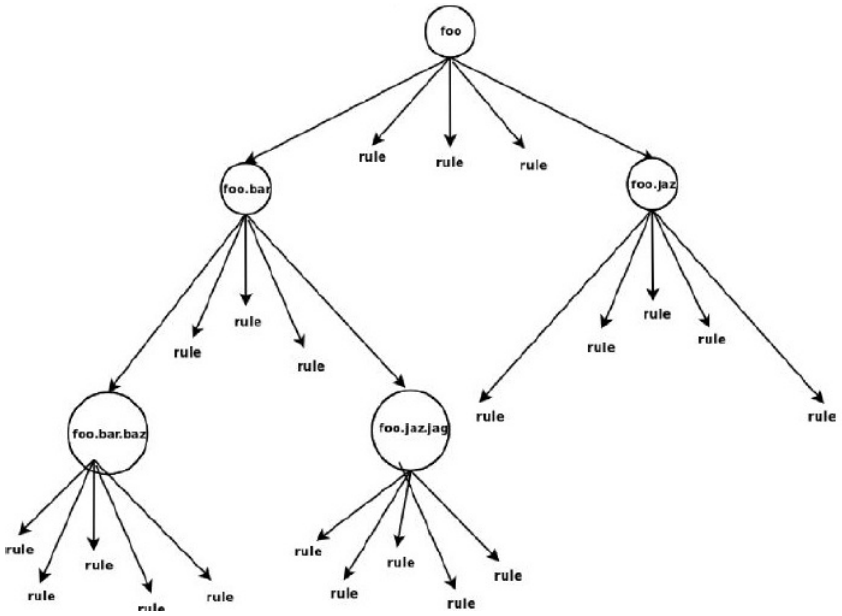


Figure 3.1: Tree representing a module hierarchy

```
Node {
Class Cl1{...}
...
Class ClN{...}

Resource Rsc1{...}
...
Resource RscN{...}

Module Mod1{... }
...
Module ModN{...} }
```

Table 3.3: Node structure

Frame: A frame defines a module's lifecycle. Different instances of the same module can be used inside different frames. A frame can be either initiated or terminated by an event (e.g. receive message event) or by the Operating System (e.g. node boot). The implication of the frame concept on the execution semantics is important.

Mandatory Frame: A mandatory frame of a module is a set of frames which comprises of the frame of the current module and the frames of all the modules which the current module uses. This is another SensL term concept that greatly affects the runtime execution.

Context: A context is the set of all the active frames of a specific event during runtime execution. An example of a context is shown the figure **??**.

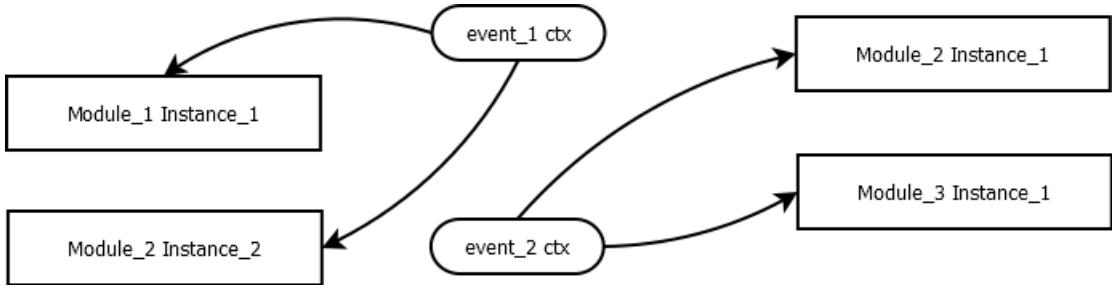


Figure 3.2: Context Concept

It is noteworthy that the context of an emitted event comprises of the context of the triggering event plus the frame of the new event. Also, the frame of the triggered event is not included in the context of the triggering event. Furthermore, each event can attach a frame inside its context only if the specific event triggers rules which belong to the frame's modules.

To summarize, the connection of the above concepts is that modules consist of a condition, zero or more properties and one or more rules. Also, frames consist of a key and one or more modules. Finally, events consist of one or more properties and one or more frames.

3.2 SensL Execution semantics

In the previous section, several concepts were introduced that have a serious impact on the runtime execution. The frame concept, the event concept, the context concept and the fact that modules and classes can be instantiated must be taken under consideration because they regulate the runtime execution. The frames instantiate modules, the context groups module instances

and the event may trigger the execution of an action of an enabled module.

To support this idea and consequently implement it, data structures (e.g. stack, queue), memory allocation and synchronous operations are needed. Thus, to implement a runtime component that handles runtime execution each platform must be able to provide the aforementioned utilities. Moreover, there are several preconditions that must be met for a rule's action part to be executed. The mandatory frame plays an important part in these preconditions and the details are provided explained below.

The mandatory frames affects states the module states as explained below:

- The mandatory frame of a module comprises of all the mandatory frames of the modules which the current module uses. So the mandatory frame is transitive.
- A mandatory frame is enabled only if all the modules which comprise it, and the modules that these modules use, and so on are enabled.
- A module is active only if its mandatory frame is enabled.

Module scope: The mandatory frames affects states the module rules as explained below:

- A rule's mandatory frame is defined as the modules mandatory frame.
- A rule is active only if its module is active.
- A rule is ready for execution only if the rule is active and enabled.

The mandatory frames also affects the events. An event is terminated if there are no ready rules in the stack. In addition, if an event is terminated and a rule triggered from the specific event is not ready for execution, the rule is not executed.

The table provided below highlights some of the aforementioned preconditions. The blank cells indicate that a definition does not exist for the SensL term, e.g. the **Ready** definition does not exist in the context of a **Module**.

SensLTerm/Definition	Rule	Module	Mandatory Frame
Enabled	Rule's Module is **Enabled** + Rule's Condition is TURE	Module's Condition is TRUE	All Modules in the current context are **enabled** + All used Modules are **Enabled**
Active	Rule's Mandatory Flame is **Enabled**	Its Mandatory Frame is **Enabled**	
Ready	Rule's is **Active** + Rule's is **Enabled**		

Table 3.4: SensLTerm / Definition Semantics

3.3 Abstract Syntax Tree (AST)

This section explains how the AST is created and then used in the MDD process. The AST is very important because it is the starting point where models become the primary artifacts that take part in building the WSN application. The EMP tools used for this purpose are described in chapter 2.

The SensL learning curve -although it is not steep due to the resemblances with object oriented languages like Java- may discourage developers to actually follow the building process proposed in thesis. To increase the ease of learning the advisor's team created a SensL editor with syntax coloring and auto-completion features by using **Xtext**. The projects that are created with this SensL editor plug-in have XText nature and consist of .snl files. These *.snl files are both text and model that conforms to the metamodel of the SensL DSL language. The SensL.ecore metamodel is automatically generated by the XText tool and it conforms to the Ecore meta-meta-model. This is important because interoperability between tools is guaranteed.

A model extracted from an .snl file has an XMI format and is called AST because is a tree representation of the abstract syntactic structure of SensL source code. An AST has many model elements due to the fact that each source code element is mapped to an Eboject. This does not make it a good candidate for transforming it into a PSM. It also does not have information for runtime execution such as the list of rules that listen to a specific event. Thus, it is a better choice to transform an AST into a simplified PIM that

contained additional information.

The complete AST is depicted in figure 3.3.

3.4 Platform Independent Model (PIM)

As stated in the previous section, the PIM is derived from the AST model. This model is independent of the underlying WSN platform and thus it plays an important role in the MDD approach. The PIM is the result of the first QVTo model transformation. Model transformations are discussed in the next chapter *Model to Model Transformation*.

The PIM model is strictly hierarchical due to the fact that the only ordering that QVTo guarantees is that of the ordered references. This means that in case of poor modeling the PIM cannot be accessed in an appropriate manner. The root object is the *Model* which contains one more or more objects of the EClass *Element*. An Element may be a *Module*, an *Event*, a *Class*, a *Node* or a *Resource*. A similar hierarchy is formed in all cases where ordering is needed. The rest of the PIM elements were created to match all the SensL terms. Each term is explicitly defined in the section 3.2: *SensL Terminology*.

Another crucial part of the PIM are *Statements* because all code blocks contain statement objects. A statement may contain a reference to the following EClasses: *If*, *For*, *While*, *Body* and *Expression*. *Method* bodies contain an ordered set of statements and with this model structure the source code information that was modeled in the AST is maintained. The ordered set of statements can efficiently be accessed using the QVTo tool.

The *Expressions* need careful treatment because they may contain various expression arguments such as constants of various types, references to model objects and have many possible operators. In addition, the PSM Expressions are modeled exactly as the PIM Expressions thus the code generation is also affected by these model elements. Each expression necessarily has a left side expression argument and an operator. A right side argument is optional, it may be used to from two sided expression of the form *expression argument* - operator - *expression argument*. Note that the expression argument may

contain an expression.

The complete PIM is depicted in figure 3.5.

3.4.1 nesC Terminology

At this point, an enumeration of and a brief explanation of the terms that belong to the target domain is helpful.

nesC program: a collection of wired components. The term **Application** is a synonym in the context of TinyOS.

Component Components are either modules or configurations. By default, components in TinyOS are singletons i.e. only one exists. There is an one-to-one mapping between file name and nesC component (also true for interfaces).

Module: A module is the main nesC source file which has two sections. First, the module{ } section consists of declarations of the from "uses interface interface_name" and "provides interface interface_name". Secondly, the implementation{ } section which consists of variables declaration, internal C functions with scope private to the component, commands -i.e. provided functions used by other components-, event handlers -i.e. functions executed upon signal reception, tasks -i.e. synchronously executed functions (scheduled with the **post** keyword).

Configuration: A configuration is a nesC source file which wires components in order to abstract and it has two sections. The configuration{ } section which consists of declarations of the from "uses interface interface_name" and "provides interface interface_name". Secondly, the implementation section which consists of wiring statements and component instantiation (using the **new** operator).

Generic Component: A generic component is an instantiable component. Thus, a generic component is reusable and prevents unnecessary code duplication. E.g. the QueueC component can be instantiated with a specific queue type and maximum size as instantiation parameters. This feature is available for TinyOS versions higher than v1.1.

Interface: An Interface describes a functional relationship between two

or more components. The interface declaration has two kinds of functions: commands and events.

Generic Interface: A generic interface is a parameterized interface. Thus it is reusable and prevents unnecessary code duplication.

Header files: A nesC header file has the same syntax and usage with C header files.

Atomic: Due to interrupts, the program execution may be preempted. To enable synchronous execution, code blocks in nesC can be declared as atomic. Such blocks are executed non-preemptively.

Async: nesC code blocks can be declared as async. Commands and events that run preemptively from interrupt handlers must be declared with the async keyword. Note that, all interrupt handlers are automatically async, and so they cannot include any sync functions in their call graph. The one and only way that an interrupt handler can execute a sync function is to post a task.

Split-phase interface: In this pattern, the request that initiates an operation completes immediately and the actual operation is executed in a callback. For example, to acquire a sensor reading with an ADC, first the software writes to registers. Then, when ADC completes, it issues an interrupt and the software retrieves the result from a data register.

Libraries: Several low (or high) level functions are already implemented as generic components. Applications usually rely on them and their provided interfaces. Examples of component-interface are: MainC → Boot, AMSend → AMSenderC, Packet → AMReceiverC, Timer⟨TMilli⟩ → TimerMilliC(), Leds → LedsC, Queue⟨t⟩ → QueueC⟨t⟩ etc.

For more formal definitions, the reader can refer to the Glossary of the document: nesC 1.3 Language Reference Manual.

Memory Allocation: Due to the new SensL operator memory management is an issue that needs to be addressed. A malloc() implementation exists in the libraries but it is problematic and may lead to reading/writing junk after a while because segmentation faults are not handled. An alternative to memory allocation is the **PoolC** component. This component does

pseudo-dynamic memory allocation. When instantiated, it allocates a space equal to an estimation of the maximum needed RAM. Then, allocation and de-allocations are performed on this memory space only.

All the aforementioned nesC terms appear in the PSM which conforms to the Ecore meta-meta-model. The statements, the expressions and the bodies are modeled in an identical way with the PIM. However, the PIM and the PSM differ greatly because they model Packet domains.

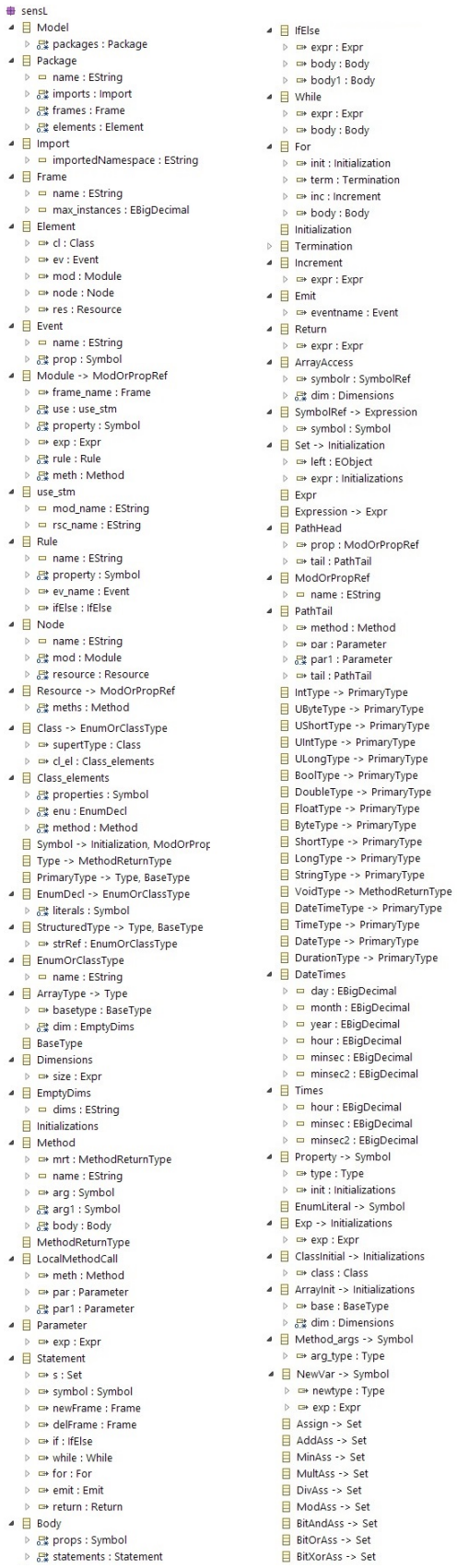
3.5 Platform Specific Model (PSM)

The purpose of the PSM implemented in this thesis is to model the concepts TinyOS platform and, once it is available, is fed as input to the code generator for the production of nesC code. This phases of the MDD process required a lot of effort in comparison with the straightforward AST to PIM mapping. This is due to the fact that the source domain is defined in terms of the SensL DSL and the target domain is defined in terms of the an existing WSN platform.

The complete PSM is depicted in figure 3.6.

3.6 Ecore Models

In the following pages the Ecore metamodels are provided.



(a) part 1 (b) part 2

Figure 3.3: Generated AST Ecore

(a) part 3

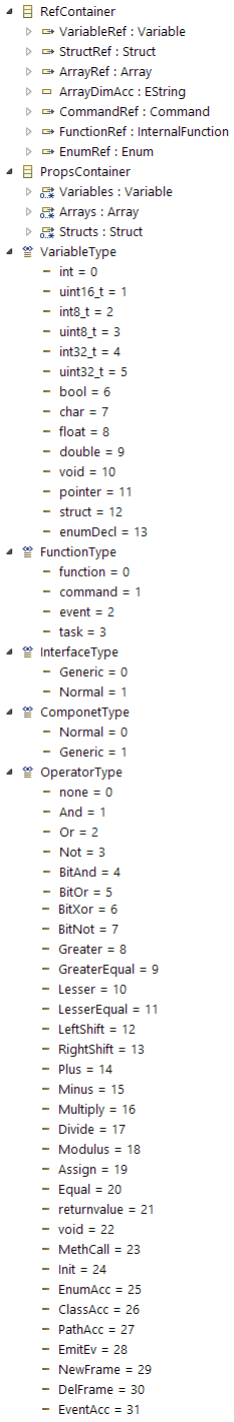
Figure 3.4: Generated AST Ecore (cont.)

(a) part 1 (b) part 2

Figure 3.5: SensL Ecore Metamodel

(a) part 1 (b) part 2

Figure 3.6: nesC Ecore Metamodel



(a) part 3

Figure 3.7: nesC Ecore Metamodel (cont.)

Chapter 4

Model to Model Transformation

4.1 SensL to Abstract Syntax Tree

As mentioned in the section 3.3, SensL files created with the SensL plug-in are also models. These models are extracted and are called ASTs. The number of the SensL files is arbitrary and thus the number of input models is not known apriori. However, the QVT tool does not support arbitrary number of input model and that is the reason why a single input model is created. The single input model conforms to the SensL.ecore metamodel and has many *Model* objects equal to the number of the SensL files. The figure 4.1 depicts a part of the SensL grammar and part of the generated SensL.ecore.

4.2 Implementation of Model Transformation

4.2.1 AST to PIM

After the extraction of the AST, it was decided not to directly transform it into the PSM. Although the AST and the PIM models are similar and model the same concepts, the PIM acts an intermediate stage. The PIM created by the AST to PIM transformation not only maintains all the AST information but is also augments the model and simplifies the consequent PIM to PSM transformation.

The AST Ecore was generated using the XText and thus it was not de-

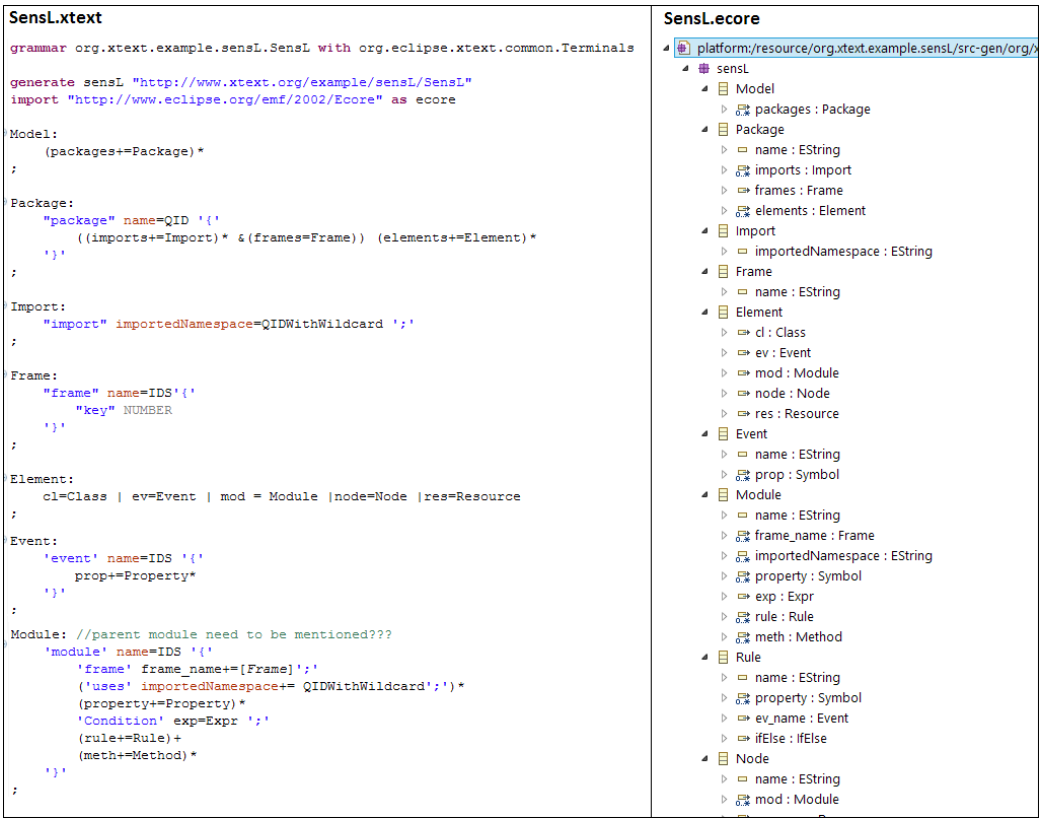


Figure 4.1: A sample of SensL.xtext in contrast with the SensL.ecore

signed for enabling an easier model transformation. Specifically, there are many EClasses for modeling expressions, i.e. one EClass for each possible expression operator. Furthermore, the modeling of property types, symbols references and assignments results to a complicated AST. However, the PIM was designed in order to reduce the aforementioned complexity and make the transition to PSM easier.

The QVT standard integrates the OCL 2.0 standard and also extends it with imperative features. The OCL as well as the extra features aided the AST to PSM transformation. These OCL operations enabled the QVTo transformation to handle all the model elements. For example, some model elements are declared EObject and the casting operation *oclAsType()* made it possible to handle them as instances of an EClass. To provide an additional example, the *oclIsTypeOf()* operation helped to determine the exact EClass

type when only the superclass type was known.

4.2.2 PIM to PSM

In order to execute the SensL application on the TinyOS platform the SensL concepts must be mapped to nesC concepts. This is done in the PIM to PSM transformation. Some mapping rules are straightforward and map a single term to another single term. However, some mapping rules required more complex transformation. These transformation rules are discussed in the following table. For SensL and nesC terms definitions and explanations refer to sections 3.1 and 3.4.1.

Table 4.1: SensL-nesC Semantics Mapping

SensL Term	**nesC Term**	**Discussion**
Property	Variable\Struct	Each SensL property is mapped to a nesC variable or a struct in case it is a complex type. The variables types are derived from a lookup table.
Module	Module	Each SensL module is mapped to a nesC module. All SensL module properties are mapped to variables and then they form a nesC struct. Also, an array of such struct is declared with size that is determined by the runtime component. This statically allocated structs are used by the runtime component.
Condition	Command	The module's condition expression is mapped to a module command. The command's arguments include a pointer to the module's struct in order to access the properties of the module's instance.

Rule	Command	The mapping is similar to the Condition to Command mapping. A command argument is needed in order to enable access to the event properties. This argument is a pointer to event struct.
Method	Command	Similar to the above mappings to a Command. It is noteworthy that the argument list is deduced by searching the method's body. For example, if the body contains references to used modules, a pointer to the used module's struct is added.
Class	Module	Similar to the module to module mapping. However, the maximum number of class instances must be known at compile time or a default maximum number is chosen.
Resource	existing nesC component	The TinyOS provides libraries that abstract the low level hardware details. Thus, a set of available resources is specified to the SensL program and according to the sensor board the proper wiring is done in the runtime configuration. This can be seen as a resource interface to resource interface that concludes with an interface wiring.
Node	Modules + Resources	For each node module the module to module mapping is invoked. The same applies for the Resources, the resource is determined from the resource to nesC component mapping. Also, the node may define constants that are used by the runtime such as the event queue size.

Package	naming convention	During the M2M phase all the package elements are renamed by adding the package name as a prefix. Due to the fact that package names are unique the produced element names are also unique.
Event	Struct	Each event property is mapped according to the property to variable mapping. The mapped variables are used to create a struct. The PSM handles these event structs differently and stores metadata such as the rule IDs that listen to this type of event. The metadata are used during the creation of the runtime component.
Frame	Struct	The frame to struct mapping is similar to event to struct mapping. The PSM also stores metadata that are used by the runtime component. However, the frames are allocated and deallocated dynamically. This is the reason that a maximum number of instances must defined. It is noteworthy that frame management is pseudo-dynamic because the maximum number of objects is statically allocated at compile time.

Chapter 5

Code generation

5.1 Implementation of M2T phase

The code generation in XPand is template-based. This means that the programmer write templates that are expanded i.e. model elements are replaced with the text that the template defines. First of all, *main* template is defined as the starting point of the transformation. Then, templates are expanded for the model elements that the current element contains. This is achieved by using the EReferences to invoke the expansion of templates from within the current template. It is noteworthy that the model must be hierarchical in order to preserve the correct ordering of the model elements.

Furthermore, to enhance the readability of the templates, the Root.xpt template is defined. This is the entry point of the Xpand tool and is used to expand the Makefile.xpt as well as the CodeGen.xpt. Thus the code-gen folder is populated with the Makefile along with the nesC files. In order to avoid a very big CodeGen template, the templates that generate the runtime component are also defined in a different file, namely the Runtime.xpt.

The code generator must produce syntactically correct code. This means that, when compiled using the existing nesC compiler, it must not produce compilation errors. The spacing between keywords, variables etc. is important but it was easy to avoid this kind of errors. Furthermore, extra attention was given to the conversion of certain model elements to string. For example, a Plus expression operator is replaced with "+' and a variable that is a struct field is accessed by using the struct name and a "." as suffix.

Another issue was the readability of the generated code. The first problem was the code structure and it was solved by invoking templates with parameters. The delegation of the current number of tab characters to all

the consequent template expansions is sufficient to maintain the code structure. For example, consider an if statement and the statements of its body. The If template passes its number of tabs to the Body template and all the body statements increase the number of tabs by one. Furthermore, newline characters had to be suppressed to avoid redundant empty lines of code.

A sample template defined in the Template.xpt is depicted in the following figure.

```
«DEFINE RuntimeHeader FOR Header»
«FILE this.name + ".h"-»
#ifndef «this.name.toUpperCase()»_H
#define «this.name.toUpperCase()»_H

«FOREACH this.HeaderEnums.sortBy(e|e.name) AS enum»«
EXPAND CodeGen::enumTmpl("") FOR enum»«ENDFOREACH»

«FOREACH this.HeaderStructs.select(s|!s.name.matches(HeaderStructs.get(0).name)).sortBy(s|s.name)
 AS struct SEPARATOR "\n"»«EXPAND CodeGen::structTmpl("") FOR struct»«ENDFOREACH»

struct {
«FOREACH HeaderStructs.get(0).Variables.sortBy(v|v.name) AS var»«""-»«
EXPAND CodeGen::varTmpl(defaultIdent()) FOR var»«ENDFOREACH-»
} «HeaderStructs.get(0).name»;

#endif //RUNTIME_H
«ENDFILE-»
«ENDDEFINE»


«DEFINE RuntimeConstants FOR Enum»
enum RuntimeConstants {
    MAX_EVENTS_IN_EQ = 10,

};
«ENDDEFINE»


«DEFINE RuntimeModule(Model model) FOR Module»
«FILE this.name + ".nc"-»
#include "RuntimeH.h"

module RuntimeC {
    uses interface Boot;
    uses interface Queue<EQEntry> as EventQueue;
```

Figure 5.1: XPand template sample

5.2 Runtime component

A very important aspect of a WSN application is the execution environment. Each underlying platform provides a unique execution model with different

advantages as well as restrictions Also, each platform supports various sensor boards a fact which affects the SensL resources. Thus, the needs of each platform must be addressed separately. To achieve this, a runtime component that implements the SensL concepts such as event emission, rule scheduling and frame allocations must implemented. In this thesis, a nesC runtime component was implemented that allows SensL applications to run on the TinyOS platform.

The SensL language has runtime semantics that are not directly reflected in the metamodels. These runtime semantics determine all aspects of the SensL application execution. The runtime component's purpose is to act as an intermediate layer between the TinyOS execution model and the SensL execution model. Given this runtime component, a nesC application can be executed on the TinyOS platform by conforming to the SensL runtime semantics. In the following paragraphs the implementation details are discussed.

First of all, the runtime component is generated with XPand along with the rest of the nesC application code files. However, it is a special hybrid case of handwritten and generated nesC code. A distinction can be made within the runtime module between the data structures which are generated and the algorithmic parts which are partly handwritten. The runtime component is modeled in the nesC metamodel with a runtime module, a runtime header and a runtime configuration. These model elements contain PSM-extracted information such as rule IDs for each SensL event, set of modules for each frame, the event queue entry type etc. The aforementioned information is used to generate the runtime nesC files. These files combined with the generated modules and interfaces form a complete nesC application.

The runtime header contains all the frame structs, the event properties structs and the event queue entry type. It also contains the enumerations that hold global constants such as the maximum allowed number of events in the event queue. This header is included in the runtime module and provides all the necessary structs that it uses. As mentioned before, the PSM contains all the necessary information for the generation of the runtime header.

The runtime module contains the global variables, arrays and structs

that are needed for the runtime execution. It contains a struct that stored the current event that is processed, the frame_masks array that is used for the allocation\deallocation of frames etc. It also uses interfaces that are needed for the sensor's startup as well as SensL module interfaces. The runtime configuration is used for the wiring of the used interfaces and resource instantiation.

The PSM-extracted information make possible the generation of commands, functions or event handlers because some variable values such as the frame id, the module id, rule id etc. are known beforehand. This allows to not store such in information in RAM. In addition, writing an event handler per event type or a createFrame() command per frame by hand would be time consuming. An alternative solution would be to create a signle hanlder with a case handling by using a switch statement or *if \else if* statements. However, the MDA paradigm does not require such effort because these code parts are generated. Every change in a SensL file, requires zero changes in the runtime template.

The runtime execution model is based on event handlers and tasks. Each SensL module can define rules that listen to the SensL *Boot* event. This is a special event type because it is the starting point of a SensL application execution. Once the sensor setup is complete the event *Boot.booted()* is called, it adds to the event queue the Boot event and posts the task *nextEvent()*. The *nextEvent()* task performs a dequeue operation and posts the proper task *process_EventType_Rules()* task. This task for each rule, checks if the rule is ready for execution and calls the corresponding command in case it is. Then it checks if the are pending events in the event queue and posts a *nextEvent()* task if there are pending events. To complete the cycle, each SensL event handlers, check if the queue was empty before was called and if it was, it post the *nextEvent()* task. The aforementioned process is depicted in the figure **??**.

The are many alternatives for implementing a rule-based system. In this thesis, the rule execution has once-off semantics. This means that each rule can be executed only once in the context of its event. Thus, each time a rule is executed a boolean *ruleExecutedOnce* is set to TRUE in order to avoid

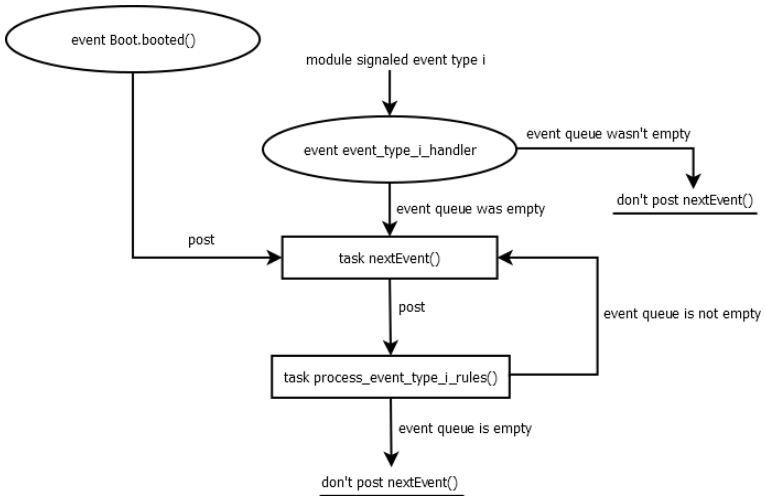


Figure 5.2: Runtime execution - Event handling

another execution even if the rule is ready for execution. To summarize, the *process_EventType_rules* task checks for each rule that listens to the event if the rule is ready for execution. If a rule is ready and the rule has not executed once then the corresponding module rule command is called.

In this paragraph the frame allocation and deallocation is discussed. A frame is allocated when the developer requests it with the *new Frame frame_name;* command. This call returns SUCCESS when the maximum number of frame instances is not reached and a new frame instance is created. The call return FAIL in two cases, either the frame is already instantiated in the context of the rule's event or the maximum number of frame instances is already reached. The frame deallocation is performed in a lazy manner due to the fact that it costs an event queue traversal. Thus when a *new Frame frame_name;* is made and the maximum number of instances is reached the function *delete_FrameType_Instances()* is called. A frame instance is deleted if no event in the queue contains the instance in its context.

Several optimizations are possible and in this paragraph discusses the compression of a context in order to reduce its size in RAM. The context is stored as an unsigned integer and the bits of this integer are indexes to module instances. The number of bits must be equal or greater that the sum of log2(max_instances + 1) for each frame. Bit-wise operations are used

to retrieve the context information e.g. to retrieve the index of a module instance which is represented by 3 bits the context is shifted to the right until these two bits are the LSBs and then an AND operation is performed with the number 7 (binary representation is 111). The figure 5.3 illustrates this idea.

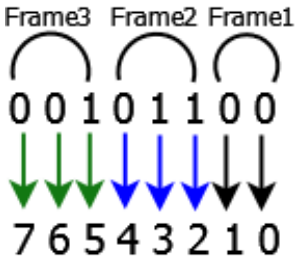


Figure 5.3: Compress event context

Another important issue is the synchronization and the avoidance of race conditions. There are only two cases in which a SensL event is added to the event queue. An event can be emitted by a hardware resource or a module's rule. Due to the fact that rules are executed synchronously within the body of a task there is no need for critical sections. The handlers are all declared implicitly as *sync* and they cannot be interrupted. Thus, the programmer does not need to worry about concurrency.

5.3 Standalone execution

To automate the process of exporting the Abstract Syntax Tree (AST), invoking the QVTo interpreter for the model transformation and invoking the XPand *.xpt a Java runable jar was created. The jar contains Java packages that utilize the tool APIs in order to programatically invoke the operations. It is also with command line arguments.

Due to the fact that some Java classes require many parameters the configuration is done via using an XML configuration file. This file is used to populate the member variables of the class *dpcm.end2ned.config.Constants*. The following figures illustrates how to set the SENSL_DIR parameter, the

folder in which the SensL text files -created using the SensL editor- are located.

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
    <!-- Convention: uri name must match
        dpcm.end2ned.config.Constants members names.
        Valid names are the following,
        SENSL_DIR;
        AST2PIM_URI;
        SENSL2NESC_URI;
        AST_URI;
        PIM_URI;
        PSM_URI;
        MODELGEN_FOLDER_URI;
        SENSL_ECORE_URI;
        PIM_ECORE_URI;
        PSM_ECORE_URI;
        CODEGEN_FOLDER_URI;
    !-->

    <uri name="SENSL_DIR">
        <protocol>file</protocol>
        <reference>folder</reference>
        <value>C:\Users\Pavlos\workspace\TestSensL\</value>
    </uri>
    <uri name="MODELGEN_FOLDER_URI">
        <protocol>file</protocol>
        <reference>model</reference>
        <value>C:\Users\Pavlos\workspace\EMF_Models\model-gen\</value>
    </uri>
```

Figure 5.4: Config parameter example

The AST mentioned in the previous paragraphs is the model representation of the SensL text. It is created from an Eclipse Plug-in created with Xtext. This is the starting point of the model-driven development because the text is parsed into a model and this model is used in the first model transformation. Additional details are explained in the chapter *Model to Model Transformation*.

To conclude this chapter, the standalone execution is important because it does not restrict the MDD solution within the Eclipse ecosystem. It also speeds up the end to end -SensL text to nesC text- process.

Chapter 6

Conclusion

6.1 Results

This thesis introduced SensL, a DSL that enables WSN application developers to write code in a high level language. Due to the execution model of this language, the programmer does not need to worry about concurrency issues. The higher abstraction level enables teams to complete separate their tasks, e.g. a domain expert working on the application layer does not need to cooperate with the person responsible for the middle-ware.

After the introduction of SensL, a model compiler was developed that translates SensL code into nesC code that can be compiled and run on a TinyOS platform by using preexisting tools. In the context of the SensL model compiler a SensL editor was implemented and a SensL PIM was written as well. These parts of the tool can remained unchanged and provide a basis for extensions e.g. support more platforms. This MDA approach offers a faster and easier development phase.

6.2 Future work

There are several important extensions to the existing work that can significantly raise the effectiveness of the solution proposed in this thesis. To begin with, SensL model compilers that support other existing platforms can be written. This will allow developers to chose the best platform for their project without worrying about the implementation details. Moreover, given SensL libraries in combination with the speedup in the development process developers will be encouraged to test new ideas.

The SensL layer in the application development process can also be used

as an intermediate layer that guarantees the separation of concerns. This implies that WSN applications can be described with high level models created with a graphical modeling tool. In this case, the models can be translated into SensL files by implementing a SensL generator tool for each graphical modeling tool. Finally, the SensL model compiler -given a platform- can generate code, ready for compilation. The aforementioned process, is a pure MDA approach that enables the creation of WSN applications by designing models. The process is depicted in figure 6.1.

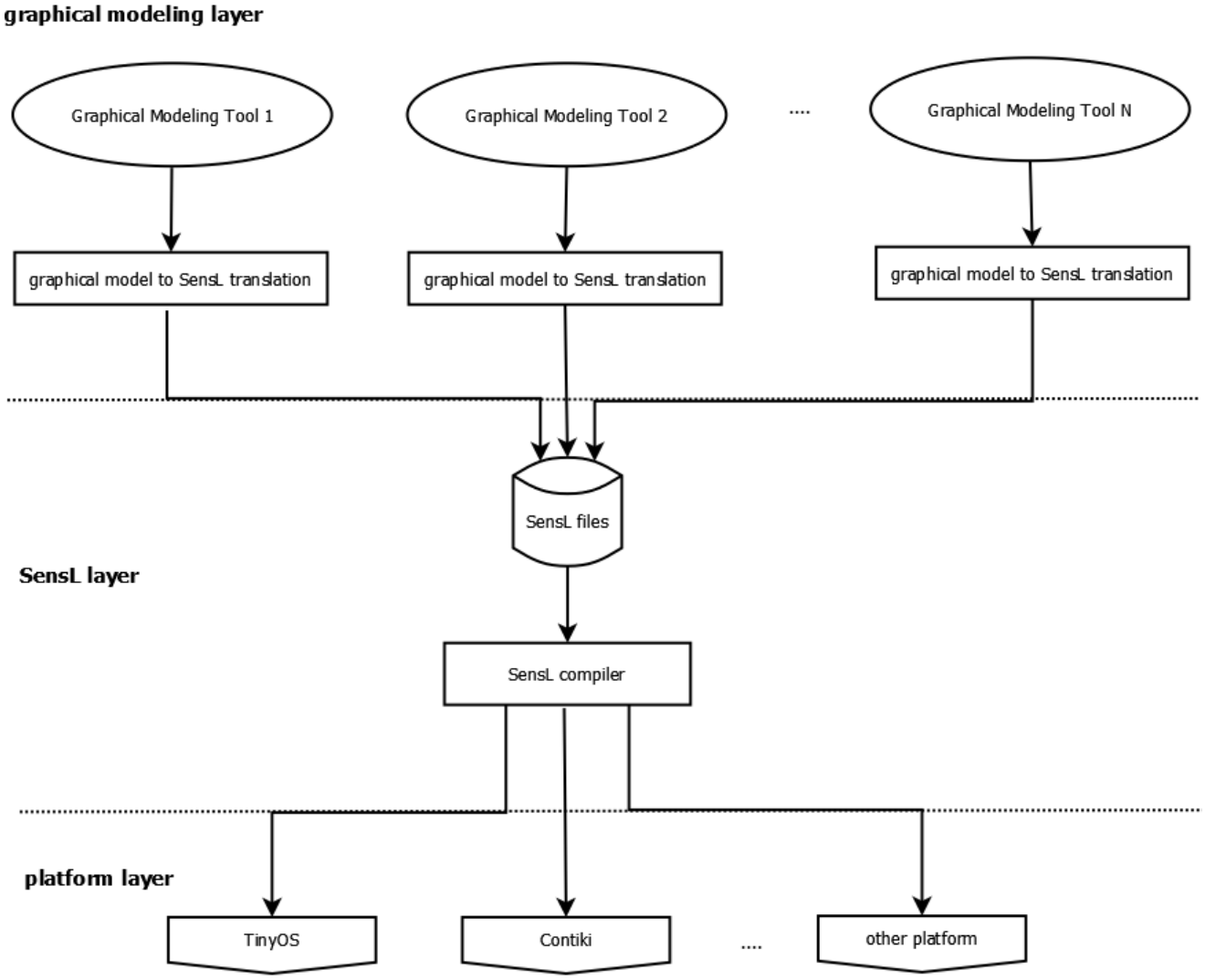


Figure 6.1: Pure model driven development

List of Figures

List of Tables

List of Abbreviations

AST	Abstract Syntax Tree
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
EMT	Eclipse Modeling Tools
EMP	Eclipse Modeling Project
M2M	Model to Model
M2T	Model to Text
MDA	Model Driven Architecture
MDD	Model Driven Development
ML	Modeling Language
MT	Model Transformation
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
QVTd	Query/View/Transformation Declarative
QVTo	Query/View/Transformation operational
PDE	Plug-in development environment
PIM	Platform Independent Model
PSM	Platform Specific Model
URI	Uniform Resource Identifier
WSN	Wireless Sensor Network

Bibliography

[1] Bran Selic. 2003. The Pragmatics of Model-Driven Development. IEEE Softw. 20, 5 (September 2003), 19-25.

[2] Rodrigues, T.; Dantas, P.; Delicato, F.C.; Pires, P.F.; Pirmez, L.; Batista, T.; Miceli, C.; Zomaya, A., "Model-Driven Development of Wireless Sensor Network Applications," Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on , vol., no., pp.11,18, 24-26 Oct. 2011.

[3] Ryo Shimizu, Kenji Tei, Yoshiaki Fukazawa, and Shinichi Honiden. 2011. Model driven development for rapid prototyping and optimization of wireless sensor network applications. In Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications (SESENA '11). ACM, New York, NY, USA, 31-36.

[4] Nguyen Xuan Thang, Michael Zapf, and Kurt Geihs. 2011. Model driven development for data-centric sensor network applications. In Proceedings of the 9th International Conference on Advances in Mobile Computing and Multimedia (MoMM '11). ACM, New York, NY, USA, 194-197.

[5] Nguyen Xuan Thang and Kurt Geihs. 2010. Model-driven development with optimization of non-functional constraints in sensor network. In Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications (SESENA '10). ACM, New York, NY, USA, 61-65.

[6] Pruet Boonma and Junichi Suzuki. 2011. Model-driven performance engineering for wireless sensor networks with feature modeling and event calculus. In Proceedings of the 3rd workshop on Biologically inspired

algorithms for distributed systems (BADS '11). ACM, New York, NY, USA, 17-24.

[7] Kai Beckmann and Marcus Thoss. 2010. A model-driven software development approach using OMG DDS for wireless sensor networks. In Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems (SEUS'10), Sang Lyul Min, Robert Pettit, Peter Puschner, and Theo Ungerer (Eds.). Springer-Verlag, Berlin, Heidelberg, 95-106.

[8] Tom Mens and Pieter Van Gorp. 2006. A Taxonomy of Model Transformation. Electron. Notes Theor. Comput. Sci. 152 (March 2006), 125-142

[9] K. Czarnecki and S. Helsen. 2006. Feature-based survey of model transformation approaches. IBM Syst. J. 45, 3 (July 2006), 621-645.

[10] Singh, Y.; Sood, M., "Models and Transformations in MDA," Computational Intelligence, Communication Systems and Networks, 2009. CICSYN '09. First International Conference on , vol., no., pp.253,258, 23-25 July 2009.

[11] Fernando Losilla, Cristina Vicente-Chicote, Brbara lvarez, Andrs Iborra, and Pedro Snchez. 2007. Wireless sensor network application development: an architecture-centric MDE approach. In Proceedings of the First European conference on Software Architecture (ECSA'07), Flavio Oquendo (Ed.). Springer-Verlag, Berlin, Heidelberg, 179-194.

[12] Luis Redondo, Rodrigo Castieira, Technical Annex v1.2, WSN Development, Planning and Commissioning and Maintenance ToolSet

[13] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks, EMF: Eclipse Modeling Framework (2008), 2nd Edition,Addison-Wesley Professional

[14] Anneke Kleppe, Jos Warmer, Wim Bast, MDA Explained: The Model Driven Architecture: Practice and Promise (2003), Addison-Wesley

[15] Martin Fowler, Domain-Specific Languages (2010), Addison-Wesley Professional

[16] David Gay, Philip Levis, David Culler, Eric Brewer, nesC 1.3 Language Reference Manual(2009)

[17] Charles Forgy, OPS5 User's Manual, Technical Report CMU-CS-81-135 (Carnegie Mellon University, 1981)

[18] Object Management Group Specifications, Meta Object Facility Core, http://www.omg.org/spec/MOF/

[19] Object Management Group Specifications, Object Constraint Language, http://www.omg.org/spec/OCL/

[20] Object Management Group Specifications, MOF Query / View / Transformation, http://www.omg.org/spec/QVT/

[21] XText Documentation, http://www.eclipse.org/Xtext/documentation/2.4.0/Documentation.pdf

[22] XPand Reference, http://www.openarchitectureware.org/pub/documentation/4.3.1/