 APPLYING MDE TO THE DEVELOPMENT OF FLEXIBLE AND REUSABLE WIRELESS SENSOR NETWORKS

CRISTINA VICENTE-CHICOTE*,†, FERNANDO LOSILLA†, BÁRBARA ÁLVAREZ†, ANDRÉS IBORRA†, PEDRO SANCHEZ‡

†División de Sistemas e Ingeniería Electrónica, Universidad Politécnica de Cartagena, Escuela Técnica Superior de Ingeniería de Telecomunicación, Edificio Antigones, Plaza del Hospital 1, 30202 Cartagena, Spain
*cristina.vicente@upct.es

Wireless Sensor Networks (WSN) are a very promising research field since they are applicable in many different areas. Current proposals for WSN system development are mainly focused on implementation issues and rarely use a Software Engineering methodology to support their development life-cycle. The Model-Driven Engineering (MDE) approach can be used as a solution to this by allowing designers to model their systems at different abstraction levels, providing them with automatic model transformations to incrementally refine abstract models into more concrete ones. In this vein, this paper presents a MDE approach to WSN application development. Three levels of abstraction have been defined which allow designers to build: (1) domain-specific models, (2) component-based architecture descriptions, and (3) platform-specific models. Automatic model transformations between these three abstraction levels have been designed and, in order to demonstrate the viability of the proposal, a real WSN application has been developed using the implemented tools.

Keywords: Model-driven engineering; component-based software architecture; domain specific languages; wireless sensor networks; Eclipse platform.

1. Introduction

Recent technological advances have led to the emergence of Wireless Sensor Networks (WSN). These systems are able to observe the physical world and to obtain useful information from it. They can process the retrieved data, make decisions on it, and carry out concrete operations on the environment.1 Nowadays, Wireless Sensor Networks find application in many different domains, such as: environmental monitoring, tele-medicine, or precision agriculture, among others.2 In 2003, the MIT's Technology Review3 published a study where WSN applications were cited as “one of the top ten technologies that will change the world”. However, current techniques for implementing this kind of systems seem to be not powerful enough to deal with their growing complexity.

Current proposals for WSN application development are mainly focused on implementation issues. Most of these systems are built from scratch following an experience-based method, which advocates selecting the most appropriate target platform first, and then the WSN domain-specific operating system (e.g. TinyOS4) and
programming language (e.g. NesC\textsuperscript{5}). The lack of a Software Engineering methodology to support the development life-cycle of these applications, commonly results in highly platform-dependent designs, difficult to maintain, scale and reuse.

The Model-Driven Engineering (MDE) approach can help reduce this dependence of the software development process on the final execution platforms\textsuperscript{6}. MDE revolves around models (defined at different levels of abstraction), and model transformations aimed at incrementally refining models into final application code. Models are defined in terms of formal meta-models (or modeling languages). These include the concepts needed to describe a system (or a set of systems) at a certain level of abstraction, and the relationships existing between them. In order to describe the model transformations needed to refine abstract models into more concrete ones, a mapping between their corresponding meta-models must be defined. Thus, applying a MDE approach requires defining both, the appropriate meta-models and the corresponding model transformations.

This paper presents a MDE approach to WSN application development aimed at improving the flexibility, scalability, maintainability, and reusability of their designs. Three meta-models have been defined at different levels of abstraction together with the corresponding model transformations. The proposed scheme allows designers to model their systems using only the WSN domain concepts included in the highest level meta-model. These initial models are then successively refined through automatic model transformations until the final application code is generated.

Before entering into details, the following section presents a motivation example based on a real WSN application for \textit{precision agriculture}, which highlights the lack of flexibility and reusability of current WSN application designs. This is followed by an outline of the research goals and process. Then, the rest of the paper is organized as follows. First, Section 2 briefly presents the platform selected to implement the tools developed as part of this work. Then, the different meta-models and model transformations implemented as part of the proposal are presented in sections 3 to 6. Section 7 reviews some related works, and section 8 presents the conclusions of the paper and some future research lines. Finally, the paper includes two appendixes: the first one presents an excerpt of one of the model transformations implemented as part of this work, and the second one, shows an excerpt of the final application code obtained as a result of applying this transformation to an example input model.

1.1. \textit{A Motivation Example}

The MITRA WSN application consists of thirty TinyOS-based nodes deployed in an almond orchard located in the semiarid region of Murcia, in the southeast of Spain. Given the shortage of water in this region, the prime objective of the system is to regulate tree irrigation according to water stress, that is, to water the trees only when it is needed. Water stress is measured using the heat pulse compensation method. This method consists of generating a heat pulse through the axial line of the tree trunk and measuring the sap temperature at two different points along this line. Similar temperatures indicate a fast sap flow and this suggests that some watering is needed.
Applying MDE to the development of flexible and reusable wireless sensor networks

The MITRA application was initially developed using a traditional approach. A new small electronic sap flow sensor was designed and the software to control both, local data processing and wireless communications was implemented in NesC, a component-based programming language for TinyOS-based WSN applications.

The resulting system was highly dependent on the TinyOS-NesC platform and on the custom sap sensor. The lack of flexibility of the design required several changes, both in hardware and in software, to cope with every small change in the application. This led us to search for a more flexible design solution, as described in the following subsection.

1.2. Research Goals and Progress

As stated before, the main goal of this research is to define a model-driven methodology for WSN application development which allows developers to build more flexible and reusable designs. This goal was initially addressed by considering the following sub-goals (labeled SGi):

SG1: Define a WSN Domain-Specific Language (DSL) which enables the description of this kind of applications at a very high level of abstraction. Models built using this WSN DSL should express all functional and non-functional system requirements, but should not include any design or implementation decisions. For this DSL to be really useful, a model editor (preferably a graphical one) must be implemented to support new Domain Model (DM) creation and validation.

SG2: Define the NesC\textsuperscript{5} meta-model from the current language specification. A graphical modeling tool for creating new NesC component-based models would be desirable, but not required since NesC models will be automatically generated from WSN DSL ones.

SG3: Define a model-to-model (M2M) transformation which maps the concepts in the WSN DSL to those in the NesC meta-model.

SG4: Define a model-to-text (M2T) transformation which automatically generates NesC code from NesC models.

SG1 and SG2 were addressed in parallel by different teams. When these sub-goals were completed (fully implemented and tested), part of the team working on the NesC meta-model started working on the M2T transformation (SG4), while the rest of the group involved in the research started working on the M2M transformation (SG3).

While the M2T transformation was successfully completed relatively quickly, the transformation between the WSN DSL and the NesC meta-model seemed a really hard problem to solve. This was mainly due to the huge semantic distance existing between the concepts in the meta-models. At this point, two options were considered: (1) to introduce some lower abstractions for system design into the DSL, or (2) to define an additional abstraction level between the two meta-models.
Option (1) meant reducing the abstraction level of the DSL and forcing developers to include design decisions within their models. In contrast, option (2) offered evident advantages and just a few drawbacks.

The use of an intermediate meta-model could help bridge the gap between WSN domain concepts and NesC primitives. As a consequence, and despite the need of defining two M2M transformations instead of one, the complexity of these transformations would be significantly lower. This intermediate meta-model could serve as an appropriate architecture description language for defining the system in terms of its components and the relationships existing between them.

With this aim in mind, a subset of the UML 2.0 meta-model, including components (for describing the system structure) and state-machine and activity diagrams (for describing component behavior), was defined to mediate between the WSN DSL and the NesC meta-model.

Figure 1 (left) outlines the elements of the proposal, that is, the set of meta-models and model-transformations defined to obtain NesC code from WSN Domain Models (DM). The intermediate architecture description language has been highlighted in order to emphasize its key role in the process.

In addition to the already mentioned benefits of including an intermediate abstraction level, it is worth noting that models defined at this level can be fully reused for different platforms. Of course, this would require defining a new PSM meta-model and the corresponding M2M transformation (from PIM to PSM) for each new target platform being considered. Furthermore, if new application domains are explored, the intermediate meta-model itself can be reused since it has been designed to be not only platform-independent but also domain-independent. This idea is illustrated in Figure 1 (right).
2. The Eclipse Platform: the Selected MDE Environment

All the tools implemented as part of this work, including all the meta-models and model transformations outlined in the previous section, have been developed using the MDE facilities provided by the Eclipse platform. This free open-source environment offers one of the most widely used implementation of the OMG standard Meta-Object Facility (MOF), called Eclipse Modeling 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. Thus many MDE-related initiatives are currently being developed around Eclipse and EMF. Among them, and directly related to the tools implemented as part of this research, it is worth mentioning the following ones:

- The Graphical Modeling Framework (GMF), which enables the implementation of graphical modeling tools from any EMF meta-model.
- The Eclipse Modeling Framework Technologies (EMFT) which enables the definition and evaluation of OCL queries and constraints on EMF models.
- The Atlas Transformation Language (ATL), which provides the standard Eclipse solution for model-to-model transformations.
- MOFScript, which supports text (and more specifically code) generation from MOF-based models.

All these Eclipse plug-ins will be briefly detailed in the sections where the tools, implemented as part of this work, are presented.

3. Defining a WSN Domain-Specific Modeling Language

The definition of a WSN domain-specific modeling language (meta-model) is aimed at helping domain experts describe their systems using only the WSN concepts they are familiar with. At this initial stage, no design decisions or concerns about the final target platform are taken into account. Conversely, models at this level must provide a clear picture of the system, defined at a high level of abstraction. For instance, a WSN domain-specific model should supply information about the overall system functionality, how this functionality is partitioned into the different nodes, how the nodes are grouped and physically distributed, what information they get from the environment, how they communicate with each other and how frequently, where the data is processed/stored (locally or remotely), and how it is presented to the user.

The WSN Domain-Specific Language (DSL) presented in this paper has been designed to help domain experts to include this information in their models. Thus, it provides the concepts most commonly used by the WSN community, together with the relationships between these concepts. As shown in Figure 2, both structural and behavioral elements have been included in the meta-model.
Fig. 2. WSN Domain Specific Language (WSN DSL).
The structure of a WSN application is defined in terms of Regions. All the nodes performing similar tasks are grouped into a logical NodeGroup, while all the NodeGroups physically deployed together are considered to belong to the same Region. NodeGroups are connected by means of WirelessLinks.

The common Behavior, shared by all the nodes belonging to the same NodeGroup, is defined in terms of FunctionalUnits. The meta-model includes an enumerated set of predefined functional units (FunctionalUnitType) which contains, among others: data management (read/write from/to sensors/ports/memory variables), communication, expression calculation, timers, etc. FunctionalUnits can be linked together by means of UnitLinks and with external Resources (Sensors, Ports, and ConfigParams) by means of ResourceLinks.

The set of all the input and output links to/from the FunctionalUnits belonging to the same NodeGroup (including both UnitLinks and ResourceLinks), define its data-flow behavior. Timers explicitly model internal node control-flow behavior, while the external synchronization mechanisms regarding inter-node message passing is only implicitly represented in the DSL models.

It is worth noting that some Resources (i.e. Sensors), are always connected as inputs to FunctionalUnits, while others (i.e. Ports and ConfigParams) can be used either as inputs or outputs, and sometimes, as both. This is why the cardinality of the inResource/outResource and the inUnit/outUnit relationships appearing in the meta-model is set to 0..1 (see Figure 2). This may sound a bit confusing since it can be interpreted as if a ResourceLink may have no input and (or) no output. However, this must be understood as the mechanism that allows designers, for instance, to link a Resource (i.e. a Sensor) which plays the role of an inResource (source of the link), with a FunctionalUnit which plays the role of an outUnit (target of the link). In this case, the ResourceLink will have neither an inUnit nor an outResource (both cardinalities shall be forced to zero).

All this confusion arises from the inherent limitation of using class diagrams to define modeling languages, since they only provide a set of syntactic rules and do not deal with semantics. In order to overcome this limitation, some meta-modeling tools, such as EMF or GMF, allow designers to annotate their meta-models with some semantic restrictions, commonly defined as OCL statements. In our case, these additional semantic constraints have been included as part of the GMF tool, implemented on top of the WSN DSL. This tool is aimed at providing WSN domain experts with an intuitive and easy to use interface for creating and validating new models which are compliant with the syntactic and semantic rules defined, on the one hand, by the WSN DSL depicted in Figure 2 and, on the other, by a set of additional OCL constraints. This GMF graphical model editor is presented in the following section.

3.1. The WSN Graphical Modeling Tool

As previously stated, all the tools implemented as part of this work have been developed using the MDE facilities provided by the Eclipse Platform. In particular, the WSN DSL
has been defined as an EMF meta-model, and the graphical modeling tool, implemented to help domain experts to create new WSN models, has been developed using the GMF Eclipse plug-in.

GMF allows designers: (1) to create a graphical representation for each domain concept appearing in a EMF meta-model, (2) to define a tool palette for creating and adding these graphical concepts to their models, and (3) to define a mapping between all the previous artifacts, i.e. meta-model concepts, their graphical representations, and the corresponding creation tools.

As stated before, GMF can be used in conjunction with the EMFT Eclipse plug-in in order to add some semantic restrictions, which can not be included in the meta-model, given the limitations of using class diagrams. These restrictions are defined as OCL constraints in the GMF mapping file, and they are validated at modeling time using the EMFT plug-in facilities. Table 1 presents one of the OCL constraints included as part of the implemented WSN DSL graphical modeling tool.

<table>
<thead>
<tr>
<th>Description</th>
<th>Conditional unit restrictions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain target element</td>
<td>FunctionalUnit</td>
</tr>
<tr>
<td>OCL constraint</td>
<td>self.type=FunctionalUnitType::CONDITIONAL_UNIT implies { self.owner.unitLinks-&gt;forall { ll</td>
</tr>
</tbody>
</table>

The constraint shown in Table 1 applies to all the FunctionalUnits defined in a given WSN DSL model, and it expresses the following semantic restriction: if the attribute type of the analyzed FunctionalUnit (self in the constraint) is set to CONDITIONAL_UNIT, then the following statement must hold: there must be at least two different UnitLinks leaving self (named ll and l2 in the constraint) and they must connect to different FunctionalUnits, that is, their targets must be different.

The MITRA system, previously described in the introduction, has been depicted using the implemented WSN graphical modeling tool. As shown in Figure 3, the model includes two Regions, R1 and R2. R1 contains two NodeGroups, one representing the MITRA nodes deployed in the almond orchard (SAP Monitoring NodeGroup), and the other representing the MITRA Irrigation Control NodeGroup (containing only one node). The second Region, R2, contains only one Sink NodeGroup with a single node.
Fig. 3. MITRA system model depicted using the WSN DSL graphical modeling tool.
The behavior of the three different NodeGroups included in the two identified Regions has been defined following this three-step process:

(i) Select the Sensors to be read from those available in the selected NodeDefinition. Note that in the WSN DSL (see Figure 2) NodeDefinitions are contained by the application rather than each NodeGroup. NodeGroups keep a reference to these definitions, making it possible to reuse them (i.e. different NodeGroups can share the same NodeDefinition).

(ii) Select and configure the activities performed by the NodeGroup from the enumerated set(FunctionalUnitType) included in the WSN DSL, and

(iii) Link all these elements together to fulfill the system functional requirements, respecting the syntactic rules defined by the meta-model and the additional OCL constraints included in the GMF application.

It is worth noting that, all the elements included in the meta-model presented in Figure 2, have been carefully selected to keep the WSN DSL as independent as possible from any specific WSN application domain and, in particular, from the precision agriculture terminology, since this DSL is aimed at modelling any kind of WSN application. This explains, why a general purpose USER_DEFINED_SENSOR was included in the meta-model (in the corresponding SensorType enumeration), instead of listing all kinds of specialized Sensors, for instance, the SAP_SENSOR used in the SAP Monitoring NodeDefinition (see Figure 3). The same applies to USER_DEFINED_PORTS and USER_DEFINED_PARAMS. It is true that this decision produces a less expressive graphical notation, since all USER_DEFINED_SENSORS appearing in the model are represented using the same icon. However, we consider it is worth keeping the WSN DSL independent from any particular application domain terminology and iconography.

The following section describes how the WSN graphical models depicted using the GMF model editor previously described, are automatically transformed into UML-based Platform-Independent Models (PIMs). The reduced set of the UML 2.0 meta-model, selected as the intermediate Architecture Description Language (ADL), and the model-to-model transformation used to refine domain-specific models to the PIM level, are presented first.

4. From Domain-Specific Models to UML-Based PIMs

As previously discussed in the introduction, a simplified version of the standard UML 2.0 meta-model was selected as the intermediate platform-independent modeling language. This intermediate abstraction level is aimed at bridging the semantic gap between the domain and the platform meta-models, reducing the complexity of the required model transformations. Thus, models created at this intermediate PIM level provide a more detailed (less abstract) description of the system under development,
Applying MDE to the development of flexible and reusable wireless sensor networks

including information about the system software architecture. PIM meta-models are often considered high-level Architectural Modeling Languages (ADL).

In this case, the proposed intermediate meta-model has been defined taking some of the elements included in the UML 2.0 meta-model and, more specifically, some of the concepts defined within the component, state-machine, and activity diagrams. Components are used for specifying the system structure, while state-machines and activity diagrams are respectively used for defining the control-flow and data-flow behavior of each component.

The structural part of meta-model, directly related to component specification, includes: SimpleComponents and ComplexComponents, Ports, PortLinks, Interfaces, and Services. The control-flow part of the meta-model (state-machine modeling) contains: States, PseudoStates (Initial, Join, and Fork), OrthogonalRegions, Events, and Transitions (including InternalTransitions and ExternalTransitions). Finally, the data-flow part of the meta-model (activity diagram modeling) contains SimpleActivities and ComplexActivities (including Conditionals and Loops), a set of ControlActivities (including Initial, Final, Fork, and Join), TimerActivities, ActivityLinks, and Actions (including AcceptEventAction and SendSignalAction).

This meta-model has also been developed using EMF although, in this case, implementing a graphical modeling tool was not necessary, since (1) models at this level are not defined by the user but automatically obtained from DSL models and, (2) EMF provides a basic reflexive model editor which is sufficient to check the correctness of these intermediate models and which allows designers to manually introduce slight variations into them to test different architectural configurations. In fact, the EMF reflective editor allows designers to change their models by adding, removing, or changing the properties of its elements, whenever the syntactic and semantic rules, imposed by the corresponding meta-model, are preserved.

The Model-to-Model (M2M) transformation required to refine DSL models into component-based PIMs was implemented using the Eclipse Atlas Transformation Language\textsuperscript{12} (ATL). This hybrid declarative and imperative language enables the definition of mappings between the concepts included in different meta-models, that is, how each concept (or set of related concepts) in the source meta-model can be transformed into a concept (or set of related concepts) in the target meta-model. One-to-one transformations are desirable but commonly they are only possible when the concepts included in the source and target meta-models are quite semantically close.

The transformation from the WSN DSL to the intermediate UML-based meta-model requires some relatively complex mappings, although some of them are also quite direct. Some of the rules, included in this ATL model transformation, are outlined next:

- Each NodeGroup is mapped to a Component.
Given the data-flow oriented behavior of WSN NodeGroups, a very simple StateMachine is associated to each Component including only an Initial PseudoState, and two States: Working and Final.

Uncoupled sets of ActivityUnits are placed into different OrthogonalRegions in the Working State, since these activities are executed in parallel. This is the most complex transformation rule since it requires detecting unconnected graphs of activities.

Each Timer FunctionalUnit is mapped to a UML TimerActivity.

All the messages sent via wireless from a NodeGroup to another are modeled as UML SendSignalActions.

Signals send by each NodeGroup to an output resource (Ports) are converted into UML SendSignalActions and, conversely, signals received from input resources (Sensors) are transformed into UML AcceptEventActions.

Each AcceptEventAction requires adding a new Event to the StateMachine and an InternalTransition in the Working State fired by this Event.

When more than one UnitLink leaves from the same FunctionalUnit (Activity), a Fork is used to simultaneously start the Activities corresponding to the UnitLink targets.

When more than one UnitLinks arrive to the same FunctionalUnit (i.e. a Timer signal and the output of another FunctionalUnit), a Join is used for synchronizing the corresponding input Activities.

The result of applying the designed ATL transformation on the initial MITRA DSL model is a UML-like platform-independent component model, which describes the system structure and behavior in terms of its components and the relationships existing between them. Although, as stated before, we have not implemented a graphical model editor for this intermediate level, the diagram presented in Figure 4 (depicted using a basic drawing tool and following the UML graphical notation), provides an easy to understand representation of part of the resulting model, and more precisely, the behavior of the SAP Monitoring Component (NodeGroup).

As it can be readily appreciated in Figure 4, the transformation has divided the Working State of the SAP Monitoring Component into two OrthogonalRegions. These regions contain the two uncoupled sets of activities identified in the NodeGroup (see Figure 3), one describing the sensing loop and another one describing how the collected data is sent to the Sink NodeGroup via wireless. Similarly, the Working State of the Sink Component is also divided into two OrthogonalRegions, one including the activities related to data collection (from SAP monitoring nodes), irrigation need estimation, and control message delivery (to the Irrigation Control NodeGroup), and another one including data retrieval and display activities.
The following section describes how these intermediate PIM models are automatically transformed into TinyOS-NesC Platform-Specific Models (PSMs), applying a new ATL model-to-model transformation.

5. From PIMs to NesC Component Models

TinyOS\(^4\) is the most widely used operating system for WSN application development and, accordingly, a wide variety of tools supporting it can be currently found in the marketplace. Among them, the solution developed by the TinyOS team is the NesC\(^5\) component-based programming language.

A NesC meta-model has been implemented, using the NesC 1.1 specification, to support the last stage of the proposed MDE approach. This meta-model comprises the following concepts: Modules (which define component implementation), Configurations (which define component groups and Wirings between them), and Interfaces (which include a list of CommandPrototypes and EventPrototypes). Modules must implement all the Commands included in the Interfaces they provide and all the Events in the Interfaces they use.

NesC applications are designed following a quite regular component pattern and thus, the rules needed to define the ATL transformation between PIMs and NesC models are not very complex. Some of these rules are outlined next:

- A different NesC model is created for each Component in the PIM.
- All TimerActivities defined in the PIM are implemented by a unique predefined NesC module, called TimerC. This Module provides a parameterized Timer interface which includes a fired event.
- Conversely, all SendSignalActions are implemented by a unique predefined GenericComm Module which provides different Interfaces for each type of Message. All these Interfaces include a SendMsg Event.
• A Main Module must be necessarily included in the design since it is invoked when the application starts. This Module is in charge of initializing the rest of Modules.
• A top level Configuration must be defined to wire all the previously defined Modules.

Figure 5 presents the NesC model associated to the SAP Monitoring Component defined in the PIM. Like in the previous case, this graphical representation has been manually depicted using a conventional drawing tool, since a graphical NesC model editor has not been implemented yet. However, the depicted model faithfully represents the model obtained as the result of applying the implemented ATL transformation.

Currently, NesC models described using the implemented NesC meta-model, only enable the description of components and their interconnections, and do not include any of the behavioral information described in the more abstract PIM models (obtained from WSN DSL models). Accordingly and, as it will be explained in the following section, in order to generate the final application code it is necessary use both, NesC models and PIMs. Some improvements in the NesC meta-model and in the ATL transformation from PIMs to NesC models are currently being developed to address this limitation, as later commented in the conclusions.

6. Code Generation

The final step of the proposed MDE methodology for WSN system development is to obtain the final application code from the previous models. In this case, the transformation is not Model-to-Model (M2M) but Model-to-Text (M2T).

M2T transformations define how model elements are converted into text patterns (in this particular case, into code primitives), while M2M transformations define meta-model
Applying MDE to the development of flexible and reusable wireless sensor networks

mappings, that is, how the elements (concepts and relationships between them) included in one of the meta-models are transformed into elements included in the other.

The final NesC M2T transformation has been implemented using the Eclipse MOFScript plug-in. This tool enables the definition of M2T transformations for MOF-based models. It provides, among others, the following facilities: model checking, parsing and querying, output file management, code completion, syntactic coloring, etc.

In order to obtain the final application code, the NesC models obtained in the previous step are not sufficient, since they do not contain complete information about component behavior. Thus, the MOFScript transformation has been designed to use both, NesC models and intermediate PIM models (containing component behavior specifications obtained from the initial DSL models). Some of the rules included in the MOFScript transformation, are outlined next:

- The NesC component model enables the generation of a list of provided and required Interfaces (provides and uses clauses, respectively) included at the beginning of the NesC file.
- The module implementation starts with a variable declaration section. One variable is defined for each <<Store>> Activity defined in the PIM with a different name, and also for each message the Component sends (receives) to (from) other components via wireless.
- For each provided Interface in the NesC model, all its Commands must be implemented. Conversely, for each required Interface, all its Events must be handled. The sequence of NesC primitives associated to these Commands and Events is extracted from the Activities defined in the PIM.

An excerpt of the implemented MOFScript transformation can be found in Appendix A, and a piece of the generated NesC code in Appendix B, both of them included at the end of the paper. The generated solution, obtained from the corresponding PIM intermediate model (see Figure 4) and the NesC component model (see Figure 5), has been successfully compiled and deployed on the MITRA WSN system, demonstrating satisfactory results.

7. Related Work

Different approaches to WSN application development can be found in the literature. Some of them offer a set of predefined components, built on top of a certain operative system, which allow designers to build new applications by configuring and combining them. For example, TinyDB\textsuperscript{14}, defines a database of TinyOS\textsuperscript{4} components that can be distributed and run in different nodes of a WSN. The information obtained by each node can be accessed from an external PC by means of SQL-like queries. Also in this line, TinyCubus\textsuperscript{15} offers a framework that enables the dynamic selection and interconnection of predefined TinyOS components. These two proposals, like most others, are focused on the implementation of platform-specific WSN applications. These approaches require a
deep knowledge of the target platform and, in most cases, the resulting designs are too platform-dependent to be reused.

Some proposals have focused their attention on the model-driven approach. GRATIS II\textsuperscript{16} offers a graphical modeling tool for designing component-based NesC applications. The underlying NesC meta-model has been defined using the Generic Modeling Environment\textsuperscript{17} (GME), a toolkit for defining domain-specific modeling and programming languages. GRATIS II offers a solution similar to the one offered by the final step of our proposal, that is, the one including our NesC meta-model and NesC model-to-code transformation. Currently we do not offer a graphical modeling tool for depicting NesC component models since these are automatically generated from the higher level meta-model, and not depicted by the user like in GRATIS II. Our NesC meta-model is also simpler than the one offered by GRATIS II, which covers a wider range of TinyOS-based target platforms and configuration modes. However, our proposal is not TinyOS (or any other platform) dependent, enabling higher level WSN application designs. Currently we support TinyOS-NesC code generation like GRATIS II, although the proposal could be easily extended to different target platforms, as stated in the introduction (see Figure 1).

The Abstract Task Graph\textsuperscript{18} (ATaG) offers a DSL for graphically describing WSN applications in terms of the tasks they must perform and the data their nodes must collect. The data-driven diagrams depicted using this DSL provide a platform-independent model of the system under development (nodes, tasks, data types, etc. are abstract to keep this platform independence). These models are similar to the activity diagrams included in our intermediate component-based architectural models. However, ATaG is architecture-independent and thus, no design information can be included within its models. Furthermore, the ATaG code generation requires the user to provide the code of each abstract task included in the model for the current target platform, offering only a semi-automated solution.

Finally, CADENA\textsuperscript{19} offers a very complete and sophisticated environment for general purpose application development. CADENA provides designers with an end-to-end model-driven environment which supports the entire application development life cycle. This tool offers, among others, a NesC plug-in which provides a graphical modeling tool (similar to the one offered by GRATIS II), and automatic NesC code generation facilities. WSN applications can also be modeled at a higher level of abstraction using CADENA general-purpose artifacts “adapted” to the WSN domain. However, adapting a general-purpose language or tool to a specific domain can be a hard work and the result will never as good (in terms of precision, efficiency, etc.) as the one obtained by defining a DSL.

8. Conclusions and Future Research

The work presented in this paper offers a new model-driven approach to WSN application development. The proposal presents a high level of abstraction using a domain-specific language, which allows designers to model their systems in a platform-independent way, thereby obtaining more flexible and reusable designs. Two additional
abstraction levels have been defined which deal with the system architecture from a platform-independent and platform-specific point of view. Automatic model transformations from the initial domain-specific models to the final application code have been implemented using the Model-Driven Engineering facilities provided by the free open-source Eclipse platform. Both the proposed approach and the tools implemented to support it have been tested on a real WSN system related to precision agriculture with successful results. The re-engineered application is fully functional and, although it is not code-optimized, the effort and the time-to-market to produce the new solution have been sensibly reduced. Actually, different solutions have been effortlessly generated thanks to the implemented infrastructure, allowing us to test new sensors and different network topologies and communication protocols.

Currently, we are working on improving the NesC meta-model and the ATL transformation from PIMs to NesC models in order to keep all the behavioral information during the whole development process. This will allow us to simplify the final code generation step using only NesC models as input.

We are also working on the integration of requirements from the very early stages of the proposed MDE approach. We have developed a Requirements Engineering Meta-Model (REMM) and a graphical requirement modeling tool aimed at defining reusable requirements catalogues. Currently we are building a catalogue of functional and non-functional WSN requirements together with a tracing tool. We hope this extension will help us, among other things, to identify and describe many of the non-functional aspects of WSAN applications, currently not very well covered in the WSAN DSL.

We are also interested in proving the benefits of our intermediate meta-model for different target platforms and domain applications. Thus, in the mid term, we plan to define new DSLs for other domains in which our research team has also some experience, such as computer vision and robotics. The wide variety of platforms currently available for this kind of systems, and the fact that some applications incorporate concepts from both domains (i.e. industrial inspection), make this future research both challenging and promising.

Acknowledgments

We would like to thank the referees for their valuable comments which helped improve this paper. This research has been funded by the Spanish CICYT project MEDWSA (TIN2006-15175-C05-02) and the Regional Government of Murcia Seneca Program (02998-PI-05).

Appendix A. Excerpt of the MOFScript M2T Transformation

This excerpt of the implemented MOFScript transformation is the module in charge of generating the code for all the NesC interfaces defined in NesC models.

textransformation NesCM2T ( in mm:"NesC" )
{ ...
module::createInterfaces() {
    self.objectsOfType(mm.Interface)->forEach(i:mm.Interface)
    {
        file f ( i.name + ".nc" )
        f.println ( "interface " + i.name + "{" )
        var count:integer
        // Adds the interface command prototypes
        i.prototypes->forEach(p:mm.CommandPrototype)
        {
            if ( p.isAsynchronous==true )
                f.print ("async command ")
            else f.print ( "command " )
            f.print ( p.returnType + " " + p.name + "{" )
            count = p.arguments.size()
            p.arguments->forEach(v:mm.Variable) {
                f.print ( v.type + " " + v.name )
                if ( count > 1 ) {
                    f.print (", ") count=count-1
                }
            }
            f.println ( ");" )
        }
        f.println ( "};" )
        // Event prototypes are similarly added
        ...
    }
}

Appendix B. Excerpt of the MOFScript M2T Transformation
This excerpt of the generated NesC code corresponds to the MitraM module defined in the NesC component model (see Figure 5).

includes MitraMsg;
module MitraM
{
    uses
    {
        interface StdControl as CommControl;
        interface StdControl as TimerControl;
        interface StdControl as ADCControl;
        interface SendMsg as Send;
        interface ADC;
        interface Timer as Timer1;
    }
}
Applying MDE to the development of flexible and reusable wireless sensor networks

interface Timer as Timer2;
interface Timer as Timer3;
interface Timer as Timer4;
}
provides
{
    interface StdControl;
}
implementation
{
    TOS_Msg msg_global;
    uint8_t counter=1;
    uint16_t reading1, reading2;
    command result_t StdControl.start()
    {
        call ADCControl.start();
        call TimerControl.start();
        call CommControl.start();
        call Timer4.start(TIMER_REPEAT, 1024*60*60*3);
        initialize_cycle();
        return SUCCESS;
    }
    ...
    event result_t Timer1.fired()
    {
        SENSOR_SEND_PULSE();
        call Timer2.start(TIMER_ONE_SHOT, 1024*60*60*3);
        return SUCCESS;
    }
    ...
    event result_t Timer4.fired()
    {
        // Builds a message containing sap measures and sends it to the sink node via wireless.
        struct MitraMsg *message =
        (struct MitraMsg *)msg_global.data;
        message->RAM1 = reading1;
        message->RAM2 = reading2;
        call Send.send(TOS_BCAST_ADDR,
                       sizeof(struct MitraMsg), &msg_global));
        return SUCCESS;
    }
}
References