Extended Abstract: Programming Wireless Sensor Networks using UML2 Activity Diagrams

Christoph Damm[1] and Gerhard Fuchs[2]

Department of Computer Science 7, University of Erlangen-Nuremberg
Martensstr. 3, 91058 Erlangen, Germany
`christoph.damm@googlemail.com`
`gerhard.fuchs@informatik.uni-erlangen.de`

***Summary*—Wireless Sensor Networks (WSNs) consist of a large amount sensor notes (spots). We address the problem of how theses spots can be controlled, so that they collaborate to fulfill a common task. UML2 Activity Diagrams (UADs) enable the user to model workflows, describing activities, in a graphical, structured and hierarchical manner. The novelty of our work is that we program WSNs using UADs. Our approach not only covers workflow description, but also action allocation. We develop a framework to design activities describing the behavior of WSNs and to execute those diagrams by spots, after an automated transformation of the model file. As a result of our first experiments, testing this framework using 12 spots, we can say that UADs can be used to program in a platform-centric as well as in an application-centric way. Additionally, we are able to adapt the behavior of a spot during runtime by reloading activities. Further research is necessary to see the full spectrum of drawbacks and benefits of our attempt.**

I. INTRODUCTION

WSNs [2] have become an important branch of research. Teething problems like routing, clustering and energy awareness in WSNs have been widely discussed and there is a trend towards discussing how to use this new technology for real applications. For us the research challange in the field of WSNs lies in the huge amount of spots (hundreds, thousands, ...), which must be coordinated. Many spots should interact and fulfill a common task. How can a programming model cope with these distributed operations?

In this extended abstract we introduce our framework that allows to program a WSN using a subset of UADs. We describe related work, basics of UADs, and our preliminary work in section II. We present important aspects of our framework in section III, section IV describes an example experiment using it. Finally section V concludes this exteded abstract, gives an brief outlook to further work, and puts some open questions.

II. RELATED WORK, BASICS AND PREVIOUS WORK

A. Programming WSNs

Sugihara and Gupta have written a detailed survey about programming models for WSNs [3]. They have introduced a taxonomy, distinguishing between an application-centric view and a platform-centric view that a programming model can take. Similar to us, they see collaboration as one important requirement for WSN applications and so for a programming model. Guerrero et al. have written a position paper [4], discussing some theoretical aspects in the field of workflow support for WSNs. To our knowledge a concrete implementation is not available. Unlike to our proposal they describe the workflows using state charts, similar to us they see the possibility to bring the programming closer to domain experts.

B. UML2 Activity Diagrams

The following subsection is based on Oestereichs book [5] that summarizes the official specification of UML2 [6], [7]. UADs describe a workflow. An activity (diagram) is defined by different kinds of nodes (action nodes, object nodes and control nodes) that are connected by object flows and control flows, symbolized by arrows. A control flow transports so called tokens, a object flow objects.

C. eXMIcutionUnit-Plugin for ROBRAIN

We gained our first experiences to programm systems using UADs during the Masters Thesis of Ipek [8]. He realized a prototype for Linux with C++ as a plugin (eXMIcutionUnit) of a multi robot programming framework called ROBRAIN [9]. As there are much more resource constraints for spots compared to the robots, there clearly was the need to create a similar (in the sense of describing workflows with UADs), but more featured and specialized framework for spots.

III. OUR FRAMEWORK

A. Avtivity-Centric View on WSNs

We aim our framework at programmers that want to code with the following view on WSNs: A spot can execute activities. If it executes an activity, it has the control about it. Its scope is limited to the workflow that is described by this activity. An action, which is included in an activity, can be allocated to a spot. The spot, executing the activity, has the control over this allocation process.

B. Components

Our framework consists of a tool for programming the UADs (`IDE`), an execution environment for UADs that runs on the spots (`CORE`), a transformation rule (`RULE`), and an access software to the WSN for the user (`ACCESS`). We use Papyrus UML 1.11.0 [10] as IDE, the rest is realized by us.

---

[1] implemented the fundament of our framework for his Master's Theses [1]
[2] corresponding author

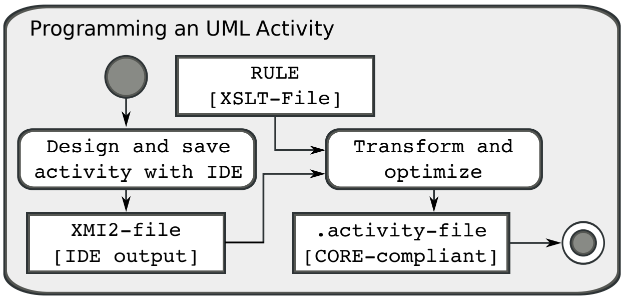

Fig. 1. Programming an UML Activity. RULE is offered by our framework.

C. Used Tools, Libraries and Technologies

CORE is realized for Sun SPOTs [11], which are programmed using Java ME [12]. We use kXml2.2.2 [13] as XML (Extensible Markup Language [14]) parser. For RULE we use XSLT (Extensible Stylesheet Language Transformation [15]). xsltproc [16] is used to convert the Papyrus UML output, which is XMI2.1 (XML Metadata Interchange [17]), to a RDF (Resource Description Framework [18]) compliant file. ACCESS is written in Java for a PC with a connected base station.

D. Features of the Current Implementation

- A programmer programs UADs by using IDE. The output of IDE is converted into a CORE-compatible syntax, using RULE.
- CORE can be pre-configured with activities, which will be loaded and parsed when CORE is started. At runtime additional activities can be added via ACCESS.
- The execution of an activity can be started by the user via ACCESS, or by the CORE of another spot.
- Status information of a spot (currently its supported activities and the battery load status) can be retrieved from CORE via ACCESS.
- CORE may execute several activities or file parsing operations simultaneously.
- A basic scope of UAD elements is supported: start nodes, final nodes, fork nodes, synchronisation nodes, decision nodes, merge nodes, guards, control flow and object flows. Furthermore implicit forks and synchronisation as well as a hierarchical structuring of activities is possible. The UADs were expanded with two stereotypes: $\ll$ *allocated* $\gg$ (to mark and describe the allocation of actions) and $\ll$ *root* $\gg$ (to mark the call of Root Activities).
- Programmers may program new RootActivities against our JAVA interface and integrate it into CORE.
- Programmers may program new AllocationProcesses against our JAVA interface and integrate it into CORE.

E. Programming UML Activities

The goal of the programming process is to gain a CORE-compliant `.activity` - file that can be interpreted by CORE (Fig. 1). First of all the programmer has to design and save the activity with IDE in an `XMI2-file`. The IDE output is transformed and optimized using RULE.

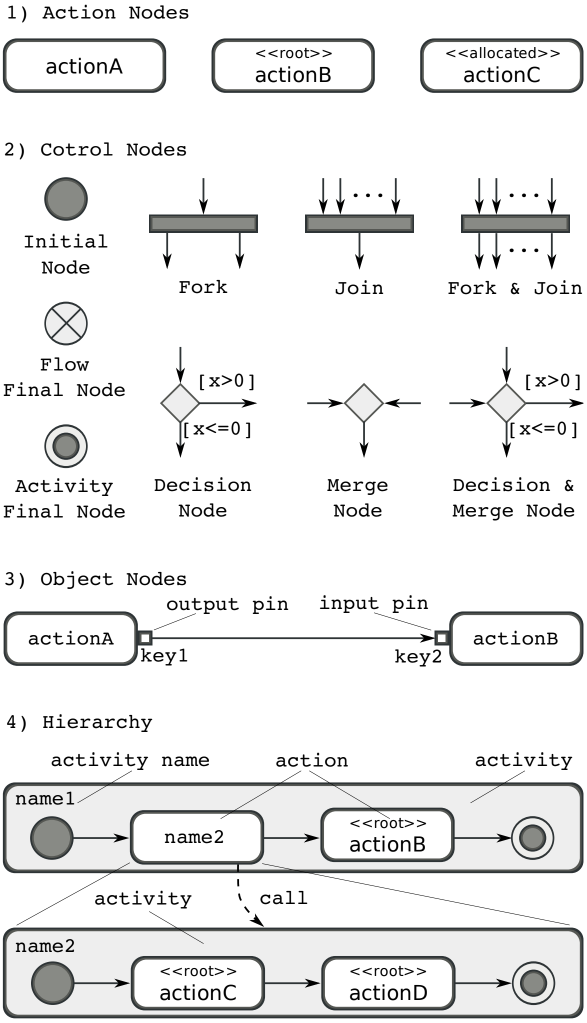

Fig. 2. Important aspects of UML2 Activity Diagrams.

F. Supported UAD elements

A programmer, who works with our framework, can currently use the elements shown in Fig. 2 to program an activity. The choosen syntax, semantik and description of the UAD elements is based on [5]–[7].

1) Action Nodes: An `action` symbolizes one step in an activity. In UML a stereotype can be used to add further information to an element. In our framework an action with $\ll$ *root* $\gg$ - stereotype symbolizes that the corresponding activity is realized in Java and not programmed using UAD. The $\ll$ *allocated* $\gg$ - stereotype shows, that the action is delegated to a spot. Our framework allows the combination of these two stereotypes.

2) Control Nodes: An `initial node` is at the start of the workflow of an activity. More than one initial nodes are possible in an activity. CORE looks for all initial notes and starts for each one a thread for the execution. A `flow final node` is at the end of a single flow. CORE stops the execution of it, the other flows are **not** stopped. An `activity final node` indicates the end of an activity. CORE stops **all** flows in the activity.

A `fork node` allows parallelism in activities. One incoming flow is immediately split in several outgoing. CORE starts for each flow a thread for the execution. A `join node` reduces parallelism and allows synchronization in activities. CORE waits for **all** incoming flows before the outgoing is started. Is is a conjunction with **and**-semantic. A `fork & join node` is a combination of a fork and a join node. CORE waits for **all** incoming flows before it starts **all** outgoing flows.

A `decision node` must have a single flow entering it, and one or more flows leaving it. At the outgoing flows conditions are annotated that specify which flow must be chosen. They are called guards and must allow a unique decision. CORE has an *GuardProcessor* that parses the annotations and allow to compare Strings, Integers and Doubles using the operators $=$, $!=$, $<$, $>$, $<=$ and $>=$. The parameter `x` must be set by an action of the activity. A `merge node` has one ore more incoming flows. CORE waits for **one** incoming flow, before the outgoing is started. This is a conjunction with **or**-semantic. A `decision & merge node` is a combination of a decision and a merge node. CORE waits for **one** incoming flow, before it uses the *GuardProcessor* to take the decision.

3) Object Nodes: An `object node` indicates that an object or a set of objects exist. CORE uses them as an incoming or outgoing parameter. IDE allows to symbolized the object node as a pin (with a square at the border of an action node). CORE uses *HashTables* for the mapping between keys and values. Per convention in our framework, the names of the input and output pins (here *key1* and *key2*) must fit to the keys in the Hashtable.

4) Hierarchy: An activity consists of single actions. If CORE detects an action it calls the activity that has the same name (here *name2*). If an action is tagged with the $\ll root \gg$ - stereotype, CORE knows, that it must call the corresponding Java-Class (here *actionA, actionB, actionC*. As we want to concentrate on the programming using UAD, we differ from the official specification, which says that a `CallBehaviourAction` indicates the call of an activity.

G. Action Allocation

We have chosen the following syntax for an instruction that can be added with the $\ll allocated \gg$ - stereotype:

$$instruction := (method : parameter : set) \rightarrow set$$

`method` specifies an AllocationProcess (entry in a list). `param` allows the modeler to specify parameters which are necessary for the allocation process. `set` is a comma-separated list of spots, from which the allocation which the allocation process must select the target set of spots. It is possible to substitute `set` with an additional instruction, so more complex allocations can be processed recursively.

IV. EXAMPLE EXPERIMENT

A. Experimental Setup

For this experiment we have build an example WSN, which consists of 12 spots. We have programmed a NetTemp and a SpotTemp activity (Fig. 3) and initially deployed SpotTemp to all spots. Afterwards we switched on the power, reset the spots, and waited about one minute[1]. We transferred NetTemp to spot a11 over the air, using the base station, started the execution and have observed the behaviour and the final state of the spots. We have made three experiments (series 1), waited about half an hour, and made additional two experiments (series 2).

B. SpotTemp and NetTemp

NetTemp and SpotTemp describe the following behaviour: The sensor network has to determine a mean value of a temperature, decide whether the result is grater or lower than 30°C, and to indicate it. SpotTemp is executed on a spot. GetTemp detects the current temperature using the sensor of the spot. The result is passed to an output-pin and if it is grater than 30°C all LEDs of the spot become red, otherwise green. NetTemp runs on spot a11 and allocates the execution of SpotTemp to four spots ($\ll allocated \gg$ - stereotype). The results are asynchronously passed to MeanValue, that is allocated to spot a09. Spot a09 start the execution not till then it has all 4 results (implicit join). If the result, calculated by MeanValue, is grater than 30°C all LEDs of spot a10 become blue, otherwise those of spot a11.

C. Observed Behavior

We have repeated the experiment 5 times. After different combinations of four spots, executing SpotTemp, turned on their (red/green) LEDs, spot a10 or a11 turned on its blue LEDs. Four times all four spots turned on their red LEDs, one time all four spots turned on their green LEDs. The green LEDs turned on at the first run of series2.

D. Interpretation and Results

As we haven't done an exact and independent measurement of the temperature, a quantitative conclusion is not possible. Qualitatively we can say, that our spots have behaved as expected. We have seen two different behaviors of the network. Both the network and the spots have made a decision and indicated it. We are able to program our WSN in a platform-centric and an application-centric way.

V. CONCLUSION AND FURTHER WORK

The huge amount of spots that must be handled in large scale WSNs is a fascinating challenge. Spots should collaborate to fulfill a common task. Based on our experiences in the field of multi robot programming, we are currently investigating how UADs can be used for programming WSNs. UADs can be used to describe workflows in a graphical, structured and hierarchical manner. Actions nodes, control notes, object

---

[1] For our experiment we used the standard communication protocol stack of the spots. This means, that we had to wait, until the spots had initialized the network.

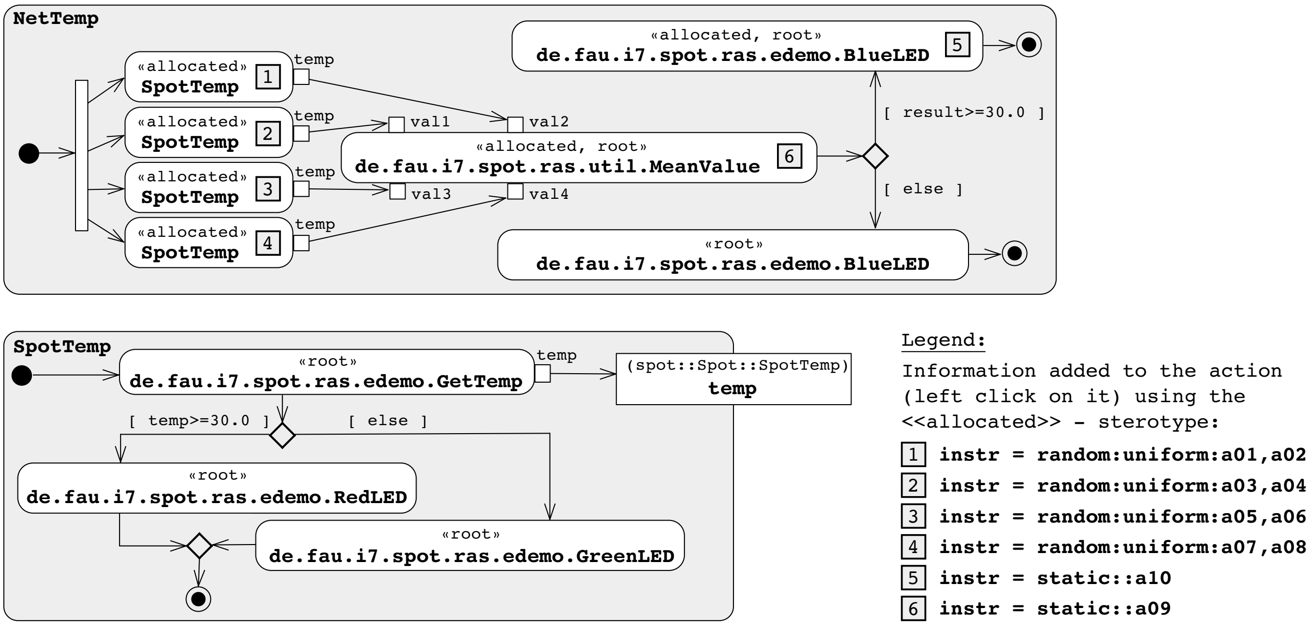

Fig. 3. Example for the programming of WSNs using UADs. NetTemp and SpotTemp are two different UADs, programmed with Papyrus UML. The images of them are the svg-export of Papyrus UML (with some typographical modifications). The diagrams are composed of the elements offered by Papyrus UML. With the exception of the boxes (1-6) the content of the diagrams correspond to the Papyrus UML view. NetTemp calls SpotTemp. NetTemp is an application-centric activity, SpotTemp a platform-centric activity.

nodes, hierarchy, parameters and stereotypes are a subset of important aspects of UADs. We use the expressiveness of them for our framework that give the user an activity centric point of view on a WSN.

Our framework uses Papyrus UML as an IDE for the design of activities, describing the behaviour of WSNs. We offer RULE for the transformation of these activities in a RDF-compliant file that can be executed by our CORE, running on the spot. Additionally we provide ACCESS to supervise, control and reprogram the spots.

First experiments, using 12 spots, show us, that our attempt can be used to program in a platform-centric as well as in an application-centric way. Additionally to workflow description, our framework currently supports static and random action allocation, and the extension of the repository of a spot during runtime, for network reprogramming.

We are currently investigating more sophisticated action allocation mechanisms and are about to increase the number of spots of our WSN. Additionally we are building robots that are controlled by the spots to gain a robot sensor network. To see all consequences of this work we have to ask amongst others: What features should / can be integrated in this framework? What is a good method and scenario for the evaluation of our framework?

REFERENCES

[1] C. Damm, “Implementierung und Bewertung eines RDF-basierten Frameworks zur Interpretierung und Ausführung von UML2-Aktivitätsdiagrammen auf Sensorknoten,” Diplomarbeit, University of Erlangen-Nuremberg, Sep. 2008.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, “Wireless sensor networks: a survey,” *Elsevier Computer Networks*, vol. 38, pp. 393–422, 2002.

[3] R. Sugihara and R. K. Gupta, “Programming models for sensor networks: A survey,” *ACM Trans. Sen. Netw.*, vol. 4, no. 2, pp. 1–29, 2008.

[4] P. Guerrero, D. Jacobi, and A. Buchmann, “Workflow Support for Wireless Sensor and Actor Networks,” in *Proc. of the 4th International Workshop on Data Management for Sensor Networks*, ser. AICPS, vol. 273. ACM, 2007, (DMSN: Vienna, AT-09; Sep. 2007).

[5] B. Oestereich, *Die UML 2.0 Kurzreferenz für die Praxis*, 4th ed. Munich, DE-BY: Oldenbourg Wissenschaftsverlag GmbH, 2005.

[6] OMG Object Management Group, “OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2,” OMG Available Specification without Change Bars formal/2007-11-04, 2007.

[7] OMG Object Management Group, “OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2,” OMG Available Specification without Change Bars formal/2007-11-02, 2007.

[8] M. Ipek, “Aufgabenbeschreibung für mobile Roboterschwärme,” Diplomarbeit, University of Erlangen-Nuremberg, May 2006.

[9] Department of Computer Science 7; University of Erlangen-Nuremberg, “ROBRAIN,” http://robrain.berlios.de/ [web: 2009/02/09].

[10] S. Gérard, “Papyrus UML,” http://www.papyrusuml.org/ [web: 2009/05/18].

[11] Sun Microsystems, “Sun SPOT,” http://www.sunspotworld.com/ [web: 2009/02/09].

[12] Sun Microsystems, “Java ME,” http://java.sun.com/javame/ [web: 2009/05/18].

[13] S. Haustein, “kXML,” http://kxml.sourceforge.net/ [web: 2009/02/09].

[14] W3C World Wide Web Consortium, “Extensible Markup Language (XML) 1.1 (Second Edition),” W3C Recommendation REC-xml11-20060816, Aug. 2006.

[15] W3C World Wide Web Consortium, “XSL Transformations (XSLT) Version 2.0,” W3C Recommendation REC-xslt20-20070123, Jan. 2007.

[16] D. Veillard, “The xsltproc tool,” http://www.xmlsoft.org/XSLT/xsltproc2.html [web: 2009/05/27].

[17] OMG Object Management Group, “MOF 2.0/XMI Mapping, Version 2.1.1,” OMG Available Specification without Change Bars formal/2007-12-01, Dec. 2007.

[18] W3C World Wide Web Consortium, “RDF Primer,” W3C Recommendation REC-rdf-primer-20040210, Feb. 2004.