UML2 Activity Diagram based Programming of Wireless Sensor Networks

Gerhard Fuchs and Reinhard German
Computer Science, Chair for Computer Networks and Communication Systems
Friedrich-Alexander-Universität Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
{gerhard.fuchs, german}@informatik.uni-erlangen.de

ABSTRACT
Wireless Sensor Networks (WSNs) consist of sensor nodes (spots). In the frame of our ongoing ACOOWEE project we assay how spots can be programmed so that they collaborate and fulfill a common task. The novelty of our work is that we see activities as scripts that can be executed by spots. Programming means to compose activity calls like bricks by specifying their sequence (workflow description) and the executing spot (action allocation).

We are developing a framework for Sun SPOTs. We use and adapt the expressiveness of UML2 Activity Diagrams (UADs) and program UADs with Papyrus UML. Our interpreter executes them after a transformation. A successful example experiment with 6 Sun SPOTs indicates us that the idea of the ACOOWEE-project could become interesting for programming distributed operation, concurrency, synchronization and data aggregation of WSNs.

Currently we are extending our framework and increasing our network. To draw conclusions for WSNs in general, further research is necessary.

Categories and Subject Descriptors
D.1.3 [Programming Techniques]: Concurrent Programming—distributed programming; D.3.2 [Programming Language]: Language Classification—uml2 activity diagrams, specialized application languages; C.2.4 [Computer - Communication Networks]: Distributed Systems—distributed applications; D.3.3 [Programming Language]: Language Constructs and Features—frameworks; D.1.7 [Programming Techniques]: Visual Programming

General Terms
Languages

Keywords
ACOOWEE, activity oriented programming, wireless sensor networks

1. INTRODUCTION
After intensive research in the field of WSNs [2] in the past WSNs shall be programmed and used for real applications. For us the research challenge in the field of WSNs lies in the coordination of a huge amount of spots (to avoid confusion, we use "spots" for sensor nodes and "nodes" in the context of UADs). In our vision many unreliable spots (hundreds, thousands, ...) are programmed to cooperate, interact and fulfill a common task. How can a programming model cope with this problem?

The Unified Modeling Language 2 (UML2) [11, 12] was standardized by the Object Management Group to allow modeling e.g. in the field of software engineering. It is widely used and so many tools are available [17]. The OMG has also standardized XMI2 (XML Metadata Interchange 2 [10]). Among others, this specification shall allow an exchange of UML2 diagrams between different tools. To adapt the syntax and semantics of the diagrams for the needs of a special domain, profiles can be specified. UML2 Activity Diagrams (UADs) are part of this standard ([12], Part II) and describe behavior. They enable the user to model workflows in a visual, structured and hierarchical manner.

The broad tool support, the standardization of the exchange data format, the adaptability of the diagrams and last but not least the possibility of visual programming are some reasons why we want to use UADs for our framework.

We gained our first experiences to program systems using UADs during the Master’s Thesis of Ipek [8]. He realized a prototype for Linux with C++ as a plugin (eXMIcutionUnit) of a multi robot programming framework called ROBRAIN [1]. Damm has adapted this concept by implementing a framework for the Sun SPOT platform [14] in his Master’s Thesis [3]. Our ongoing ACOOWEE project is based on the results of these thesis, particularly on the implementation of Damm [4].

Sugihara and Gupta have written a detailed survey about programming models for WSNs [13]. None of the listed models uses UADs. Guerrero et al. have written a position paper [7] 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.

The remaining paper is organized as follows: In section 2 we introduce the idea and the goals of our ongoing ACOOWEE project. Section 3 introduces our ACOOWEE framework. To illustrate and test our attempt we present an example experiment in section 4. Finally section 5 concludes this paper and gives a brief outlook to our further work.

http://doi.acm.org/10.1145/1809111.1809116
2. THE ACOOWEE PROJECT

2.1 Basic Idea

In the ACOOWEE ("Activity Oriented Programming of Wireless Sensor Networks") project we pursue the idea that an activity is a script that can be executed by an interpreter running on a spot. Activities can have an input to receive data and an output to deliver data (fig. 1.1). Data are passed between activities, by connecting an output with an input (fig. 1.2). An initiator starts the execution of an activity by sending a request that contains the activity name and the input data (fig. 1.3). As a result the spot processes the request and returns the output data. The initiator can be the spot itself (local action call) or RPC like (remote procedure call) another spot or the user.

The start of an activity is an action which we see as a brick. Fig. 2.1 shows the principle of Activity Oriented Programming of WSNs. Bricks are composed to new activities by specifying their sequence (workflow description) and their fulfilling spots (action allocation). This recursive definition of an activity is finished by RootActivities. A typical example for this type of activity is the read-out of a sensor value. RootActivities of a spot can be compared to the instruction set of a microprocessor. They have to be written in the programming language which is offered by the spot.

Action allocation can be static or dynamic. Static means, that the programmer specifies an exact mapping between the activity and the executing spot (e.g. activity A runs on spot 1). Dynamic means that the programmer specifies a rule how the executing spot must be selected during runtime (e.g. select randomly one spot from all local neighbors that are reachable via one hop). If all activities are executed locally the behavior of one spot is programmed, if actions are allocated to other spots the behavior of (a part of) the WSN. An interpreter that executes an activity on a spot has to realize the specified workflow and the action allocations.

A spot can store activities and RootActivities in its repository (fig. 2.2). The content of the repository are the spot’s capabilities. They can be described by a profile and adapted by adding, removing or replacing activities. Similar to the execution of an activity the adaption can be initiated by the spot itself and RPC like by another spot or by the user.

2.2 State of the Art, Activities and Goals

We have built a prototypical framework to visually program networks consisting of Sun SPOTs using a subset of UAD elements.

We can rudimentary describe work flows and data flows of one Sun SPOT as well as of the network using action allocation. We are currently focusing our research on languages for specifying the workflow and the action allocation. For the workflow description we assay how the syntax and semantics of UADs can be used or adapted to act as a glue for the composition of the activities. For the action allocation we are working on a own syntax and semantics. In the field of action allocation we are integrating dynamic and exploring action allocation mechanisms based on "local neighbors", "energy awareness" and "probabilistic methods". We are studying methodologies for dynamic reprogramming and code adaption. In addition to the concept presented in [5] we are enabling our Sun SPOTs to exchange code between each other. We are building a network of 91 Sun SPOTs (fig. 3). To gain mobility and heterogeneity we are developing and assembling robots that are controlled by Sun SPOTs.

We want to extend our framework with further UAD elements (e.g. signaling) and allow group allocation, considering different aggregation methods. Our goal is to abstract and generalize our ideas and experiences gained form the development of our framework and draw conclusions for WSNs in general.

3. THE ACOOWEE FRAMEWORK

3.1 Components and Features

Our framework consists of a tool for programming UADs (IDE), an interpreter for UADs that runs on the Sun SPOTs (CORE), a transformation rule (RULE), and an access software to the network for the user (ACCESS) that runs on a
host with a connected base. We use Papyrus UML 1.11.0 [6] as IDE, the rest is realized by us. CORE is realized with Java ME for Sun SPOTs. ACCESS is written for a PC in JAVA.

For RULE we use XSLT (Extensible Stylesheet Language Transformation [16]) in conjunction with xsltproc [15].

A programmer visually programs UADs by using IDE (fig. 4). At this he can concentrate on the programming area, the element area and the properties area during a programming process. The programmer assembles an UAD in the programming area via drag-and-drop using the UML Links and UML Elements from the element area. A mouse click (left button) on an element in the programming area opens the properties area. Here the programmer can add stereotypes and so information for the action allocation. The example shows the properties of the labeled *edemo.whiteLED* RootActivity. The output of IDE is converted into a CORE-compatible syntax, using RULE.

We offer a Java-interface for the programming of RootActivities. CORE can instantiate the realizing classes, pass the input data, start/stop the execution and fetch the output data. This activities are the link between the Java ME api of the Sun SPOTs and the UADs.

CORE can be pre-configured with UADs and RootActivities which will be loaded and parsed when CORE is started. At runtime additional UADs can be added via ACCESS. The execution of an UAD can be started by the local CORE, the CORE of another Sun SPOT, or by the user via ACCESS. Status information of a Sun SPOT (currently its supported activities and the battery load status) can be retrieved from CORE via ACCESS. CORE may execute several activities simultaneously. A basic scope of UAD elements is supported.

We have implemented a prototypical, proprietary RPC like communication protocol. It is used for the communication between two COREs as well as between ACCESS and CORE.

At its current stage of expansion CORE has 1821 loc ("lines of code", counted with cloc1), the RootActivities 370 loc, the communication part 298 loc, and ACCESS 208 loc.

---


---

**3.2 Supported UAD elements**

A programmer, who works with our framework, can currently use the elements shown in fig. 5 to program an UAD. The chosen syntax, semantics and description of the UAD elements is based on [9, 12].

1) **Action Nodes.**

An action symbolizes one brick in an UAD. In UML a stereotype can be used to add further information to an element. In our framework an action with ≪ root ≫ - stereotype symbolizes that the corresponding activity is a RootActivity and so realized in Java, not programmed using UADs. The ≪ allocated ≫ - 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 UAD. More than one initial nodes are possible in an UAD. 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 UAD. CORE stops all flows in the UAD.

A fork node allows parallelism in UADs. 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 UADs. CORE waits for all incoming flows before the outgoing is started. It is a conjunction with and-semantics. 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 UAD. 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-semantics. 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 data is passed between two activities. CORE uses them as an incoming or outgoing parameter. IDE symbolizes 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 Hash-table. A data flow between two Sun SPOTs is programmed by allocating actionA to another Sun SPOT as actionB.

4) Hierarchy.

An UAD consists of single actions. If CORE detects an action it calls the UAD that has the same name (here name2). If an action is tagged with the ≪ root ≫ - 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 UAD.

3.3 Action Allocation

During the execution of an activity, CORE works off the actions that are specified in the workflow. At this it always checks whether the programmer has added an ≪ allocated ≫ - stereotype to the action. If the ≪ allocated ≫ - stereotype is not added, CORE locally starts the execution. Otherwise it utilizes the added information, determines the executing Sun SPOT and delegates the execution RPC like to it. We have chosen the following syntax for an instruction that can be added with the ≪ allocated ≫ - stereotype:

\[\text{instruction} := \text{method} : \text{parameters} : \text{set} \to \text{set}\]

Example:

\[\text{instruction} := \text{random} : \text{uniform} : a01, a02\]

method specifies the allocation method (e.g. "random" means "Choose randomly ..."), CORE has a look-up table which specifies its reaction, parameters allows the programmer to specify parameters which are necessary for the allocation method (e.g. "uniform" means "... considering an uniform probability distribution ..."), set is a comma-separated list of spots, from which the allocation method must select the target set of spots (e.g. "a01,a02" means "... from the Sun SPOTs with the name a01 or a02").

It is possible to leave set unspecified, or to substitute set with an additional instruction. So more complex allocations can be recursively processed. If no set is specified the allocation method must generate a set of Sun SPOTs. If set is substituted by another instruction, CORE uses the result as input for the other instruction. Example:

\[\text{instruction} := \text{random} : \text{uniform} : \text{local} : \text{hops} = 2 :\]

means "Choose randomly considering an uniform probability distribution from the Sun SPOTs that are your local neighbors reachable via 2 hops." At the "random"-instruction set is substituted by the "local"-instruction. As the "local"-instruction has no set specified, it must generate a list of Sun SPOTs.

The presented syntax allows to return more than one Sun SPOT (set) for the execution of an activity. As we are currently not supporting group allocation, we are simply choosing the first Sun SPOT. We are confident the group allocation is possible in future.

4. EXAMPLE EXPERIMENT

With this experiment we want to illustrate the usage of the ACOOWEE framework, test it, and give an example for programing the behavior of one Sun SPOT / of the network. Our example includes all elements which we support at the moment.

4.1 Experimental Setup

For this experiment we use an example network which consists of 5 Sun SPOTs (ax). One further Sun SPOT is connected to a PC and is used as a base station to access the network. We have programmed the RootActivities WhiteLED, RedLED, GreenLED, BlinkLED, NoLED, MeanValue, Get-
Temp, and $\text{Wait5Sec}$ against our Java-interface. Additionally, we have programmed a NetTemp6 UAD and a SpotTemp UAD (subsection 4.2) using the IDE (fig. 4) and transformed the resulting output using RULE. We initially deploy the RootActivities and SpotTemp to a01-a05. Afterwards we switch on the power, reset the Sun SPOTs, and wait a few seconds. We transfer NetTemp6 to CORE of a05 over the air. At this we use ACCESS that runs on the PC with the connected base station. Afterwards, we start the execution via ACCESS and observe the behavior and the final state of the network.

The size of the SpotTemp UAD is 2.8 K, of the NetTemp6 UAD 5.1 K, of the suite that is deployed on the Sun SPOTs 86 K, and of the ACCESS-jar 11 K.

4.2 SpotTemp and NetTemp6

SpotTemp (fig. 6.1) and NetTemp6 (fig. 6.2) describe the following behavior: The network has to determine a mean value of a temperature, decide whether the result is grater or lower than 30° C, and indicate it.

SpotTemp is composed of RootActivities. It dose not use the $\ll\text{allocated}\gg$ - stereotype, so it is executed on a single Sun SPOT. After the start of the activity (a), GetTemp (b) causes the Sun SPOT to detect the current temperature. As the result is passed to the output-pin (c), it can be used in NetTemp6. Next, the Sun SPOT must take a decision (d). If the temperature is grater than 30° C all LEDs of the Sun SPOT become red (e), otherwise green (f). The flow merges (g) and the Sun SPOT waits 5 seconds (h). This allows the user to see the decision before the LEDs are switched off (i) and the execution is stopped (j).

NetTemp6 runs on a05. After the start of the execution (k), a5 starts four concurrent flows (l). It randomly allocates the execution $\ll\text{allocated}\gg$ - stereotype of SpotTemp to a01 or a02 (m), and statically to a03 (n). The measured temperatures are asynchronously passed to a05. If a05 has both temperatures (implicit join), it starts the execution of the mean value on a04 (o). At this, the temperatures are passed as parameters (val1, val2). The calculated result is returned to a05, so it can offer netTemp as an output (p).

Parallel to this flow, a05 advises a04 to switch on its white LEDs (q). When a04 has finished its activities (r), a05 takes a decision (s). If the result of the calculation of the mean value is grater than 30° C, it advises a04 to switch on its red LEDs (t), otherwise its green LEDs (u). The flow merges (v) and a05 waits for 5 seconds (w). This allows the user to see the result of the decision before the LEDs are switched off (x) and the execution is stopped (y).

During the whole time the blinking of a05’s LEDs (z) indicates the execution of NetTemp6. We have programmed BlinkLED as an infinite activity. a05 only stops the blinking when the NetTemp6 activity is finished (y). For the sake of completeness, we have added the Flow Final Node (aa). It has no influence on the execution.

4.3 Observable Behavior

After the start of the execution of NetTemp6 via ACCESS, a05 starts blinking its LEDs, a04 switches on its white LEDs, a03 switches on its green/red LEDs, and a01 or a02 switches on its green/red LEDs. This state keeps about 5 seconds, before a1, a2, a3 switch off their LEDs and a04 switch on its green/red LEDs. After additional 5 seconds all Sun SPOTs switch off their LEDs and the netTemp-parameter is returned to ACCESS. The usage of a heat gun changes the behavior of the Sun SPOTs and the network.

4.4 Discussion

We have programmed one Sun SPOT as well as the network. We see two different behaviors of the network. Both the network and the Sun SPOT makes a decision and indicate it. Our Sun SPOTs behave as expected.

In the experiment we direct the work flow and data flow to 4 different Sun SPOTs. This is an important aspect for distributed operation. SpotTemp is started on 2 Sun SPOTs, there are 4 concurrent flows. We have specified concurrency, synchronization, and data aggregation (MeanValue). In the experiment we add the NetTemp6 UAD to a1 during runtime. In conjunction with profiling techniques we see the possibility for dynamic network reprogramming.

5. CONCLUSIONS AND FURTHER WORK

Inspired from our experiences in the field of multi robot programming, we have started to investigate how UADs can be used or adapted to program a huge amount of unreliable spots of a WSN so that they cooperate and fulfill a common task.

In this paper we have presented our ongoing ACOOWEE project: We see an activity as a script that can be executed by a spot and the call of an activity as a brick. In this context programming means to compose these bricks by specifying their sequence (workflow description) and the executing spot (action allocation).

We are setting up a laboratory consisting of Sun SPOTs. As we have extended the expressiveness of UADs by an $\ll\text{allocated}\gg$ – stereotype (to specify allocation rules) the user of our framework can not only program the behavior of one Sun SPOT, but also the behavior of the network. The UADs are visually programmed with Papyrus UML. Afterwards, the XMI output is transformed by our rule to a data format that can be executed by our interpreter. A simple experiment gives us a hint that our attempt could be interesting for programming distributed operation, concurrency, synchronization, and data aggregation of WSNs.

We are currently supporting only a subset of UAD elements. To see all benefits and drawbacks of UAD based programming of WSN, we have to extend our framework and generalize our attempt.

6. REFERENCES


Figure 6: Example for the programming of WSNs using UADs. NetTemp6 and SpotTemp are two different UADs which are programmed with Papyrus UML. The diagrams are composed of the elements offered by Papyrus UML. NetTemp6 programs the behavior of the network, SpotTemp of a single Sun SPOT.


