RULE-ORIENTED PROGRAMMING FOR WIRELESS SENSOR NETWORKS

Kirsten Terfloth
Freie Universität Berlin
Takustr. 9
14195 Berlin, Germany
terfloth@inf.fu-berlin.de

Georg Wittenburg
Freie Universität Berlin
Takustr. 9.
14195 Berlin, Germany
wittenbu@inf.fu-berlin.de

Jochen Schiller
Freie Universität Berlin
Takustr. 9
14195 Berlin, Germany
schiller@inf.fu-berlin.de

Abstract Data-centric, distributed programming for embedded systems with harsh resource constraints poses a heavy burden upon a developer. In this paper, we describe how rule-based programming can alleviate these problems by combining middleware and application at the programming level.

We describe in detail the programming primitives and the implementation of the FACTS middleware architecture. Based on statistics derived from three representative tasks specific to wireless sensor networks, we illustrate how our approach allows for aggressive optimization as well as writing expressive application-level code. We summarize our experience by proposing several rule-oriented programming patterns.

Keywords: Wireless Sensor Networks, Programming Abstraction, Middleware Framework, Rule-Oriented Programming Patterns

1. Introduction

As wireless sensor networks become ever more popular even beyond the scientific scope, the need for sophisticated programming tools is accentuated more often. Support may be provided in a variety of ways, each having different values for different tasks to accomplish. While hiding network interaction from the developer may be beneficial for some applications, one will in turn have to sacrifice deliberate control mechanisms for energy consumption. In this paper we will show how to efficiently program with FACTS, a rule-based runtime environment targeted at wireless sensor networks. The key to our approach is to strictly stick to a data-centric paradigm, optimize local node behavior and hide concurrency issues and manual stack management from the programmer. Combining the advantages of a runtime environment with its sandboxed execution and the possibility to obtain very dense bytecode, FACTS offers a suitable programming abstraction for wireless sensor networks.

2. FACTS Concepts and Language

The three basic concepts of the FACTS middleware framework are *facts*, the key abstraction for data within the sensor network, *rules*, providing means to manipulate, process and communicate data, and *functions*, which offer an interface to the underlying hardware and software stack of the target platform. We will first briefly introduce the architectural model before we give a formal specification of the different building blocks of the Ruleset Definition Language (RDL).

2.1 Architectural Model

The architectural model as depicted in Figure 1 gives the programmer a very simple, yet powerful tool to design distributed algorithms. The core of the FACTS middleware is the rule engine, the central scheduling entity on the nodes. Whenever a fact is either received over the radio interface targeted at the runtime environment, or an event has been detected, the rule engine will be triggered to run. According to their priority ordering, all rules currently deployed on the node will be checked against the fact repository. If all conditions a rule specifies are met and at least one of the facts involved in the satisfaction has been modified, having thus changed its state during the last run or having been newly added to the repository, the rule will fire and its statements will be executed. Concurrency issues and memory management are shielded completely from the programmer. In order to enable access to firmware

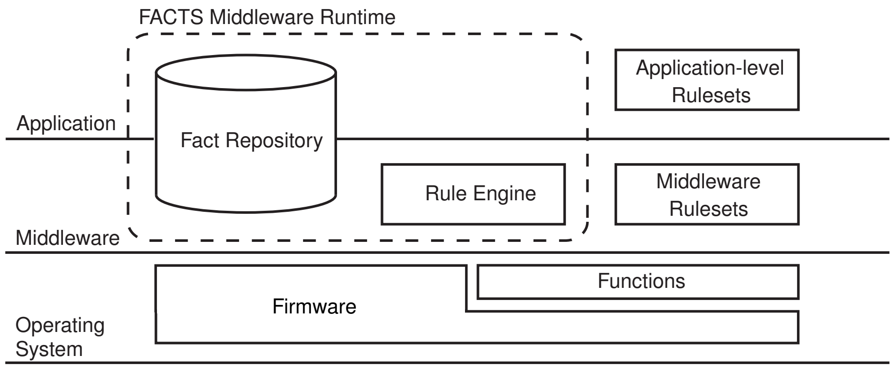

Figure 1. Interaction between the components of the FACTS middleware architecture.

functions where necessary, a special call statement can be invoked, e.g. to have read access to sensors or toggle LEDs on the sensor board.

2.2 Language Concepts

Let $F = \{f_1, f_2, ..., f_n\}, F \subset F^*$ denote the set of facts stored in the fact repository, where F^* corresponds to the set of all well-formed facts. Each fact may have a set of associated properties
$P = \{f_1.p_1, f_1.p_2, ..., f_1.p_a, ..., f_n.p_b\}$ within the domain of all well defined properties P^*, $P \subset P^*$. Let $R = \{r_1, r_2, ..., r_m\}$ be an arbitrary set of rules with a priority ordering $Prio = \{r_i > r_j | \text{rule } r_i \text{ has precedence over } r_j\}$. The definition of priorities is part of the rule syntax and therefore obligatory to the programmer. No guarantees will be given on the order of execution of rules with the same priority assigned.
A rule r_i is composed of a set of conditions $C_i = \{ r_i.c_1, ..., r_i.c_d\}$ stating when a rule fires, and a set of statements $S_i = \{r_i.s_1, ..., r_i.s_e\}$ specifying what action will be taken. A condition can be of two possible types:

- $C_{ex} = \{\langle Ex, f \rangle \mid f \in F^*\}$, the `exists` condition, allows to check whether a certain type of fact is currently stored within the fact repository.

- $C_{ev} = \{\langle Ev, f.p\rangle \mid f.p \in P^*\}$, the `eval` condition, can be used to investigate the properties of facts against thresholds or in relation to other facts in the repository.

The statements of a rule r_i will only be triggered if $\forall_{j=1}^{d} r_i c_j = true$, with d being the number of associated conditions. Furthermore, at least one of the facts involved in condition satisfaction has to be newly added or updated during the last run of the rule engine for rule execution. Whenever no modified facts are present the fact repository, the sensor node can save energy by switching to low-power mode.

While we currently assume a static set of rules deployed on the nodes, facts are dynamic and may change at runtime. Let S denote the set of available fact modification statements:

- $S_d = \{\langle D, f\rangle \mid f \epsilon F^*\}$ is a `define` statement, stating that a new fact with or without specified properties is being added to the fact repository.
- $S_r = \{\langle R, f\rangle \mid f \epsilon F\}$ is a `retract` statement to be invoked if fact(s) shall be deleted from the fact repository.
- $S_u = \{\langle U, f.p\rangle \mid f.p \epsilon P^*\}$ denotes the group of operations that update specified facts. This can either be a `set`, that operates on properties of facts, or a `flush` statement which will disable the specified fact from triggering a rule in the next run of the rule engine.
- $S_s = \{\langle S, f\rangle \mid f \epsilon F\}$ `send`s addresses fact(s) over the radio interface. Both, unicast and broadcast messages are possible.
- $S_c = \{\langle C_i, f.p\rangle \mid i \epsilon I, f.p \epsilon P\}$ with I being the set of identifiers of functions defined in the firmware interface. This statement provides supervised access to the underlying software stack, possibly enabling resource-intense computations to be implemented in native code.

In rule-based programming, one tends to address a subset of facts for processing or rule triggering more than once. In order to make the code less verbose, we introduced the notion of *slots*. Slots are named filters for facts, which will be introduced based on an example in Section 1.3. Another crucial part of RDL are *rulesets*, specifying a set of rules interacting with each other, analogous to a compilation unit in traditional programming languages. Rulesets may therefore be used to encapsulate services like routing, node self-inspection or application semantics.

2.3 Bytecode Structure and Optimization

Efficient use of available memory is a crucial aspect of programming wireless sensor nodes. In order to illustrate to which extend our approach satisfies this requirement, we will now describe how the bytecode images of compiled rulesets are structured and in in which way they can be optimized for size.

	# Rules	LOC	Size		
			unoptimized	optimized	% saved
Directed Diffusion	7	76	2,204 B	916 B	58.4 %
Generic Role Assignment	14	196	6,806 B	1,950 B	71.3 %
Coverage	7	99	5,392 B	1,064 B	80.3 %

Table 1. Code size statistics of different rulesets.

Within the bytecode, data structures corresponding to the same programming primitives are organized as arrays to save the structural overhead of linked lists. The arrays for facts and properties are special as they are the only arrays that will change in size at runtime. As far as code organization is concerned, the sequence of statements within each rule is the only feature that rulesets share with traditional imperative programs. Apart from this, there are no constraints on how a ruleset can be laid out in the bytecode, thus allowing for aggressive optimization in order to reduce the size of the bytecode image. The ruleset compiler parses rulesets implemented in RDL and generates a tree-like data structure to be used in the bytecode. The key to optimizing this data structure is to find identical subtrees, merge them into one, and adjust pointers in the remaining data structure accordingly.

In order to examine the impact of bytecode optimization in detail, we have collected results from three major rulesets: Directed Diffusion [5], Generic Role Assignment (GRA) [3] and a distributed network coverage algorithm. The code size statistics of these rulesets are given in Table 1.

Figure 2 shows the percentage of memory saved through bytecode optimization for each type of programming construct. The total savings achieved for each ruleset of well over 50% may seem unrealistic at first glance. Examining the optimization in detail, we find that the savings are to be attributed mostly to redundant expressions, variables and slots. In fact, these three primitives are heavily used when implementing rulesets. However when looking at usage patterns, it becomes obvious that there are only very few different instances of these elements, which are repeated very often. In contrast, rules, facts and properties have not been optimized at all because there are no redundant rules in any of the rulesets examined and facts and properties cannot be optimized at all even if they were identical. Contrary to other elements of a ruleset, facts and properties may be modified at runtime, and hence even properties that are identical at compile-time may be assigned different values at runtime, thus needing separate memory to store them.

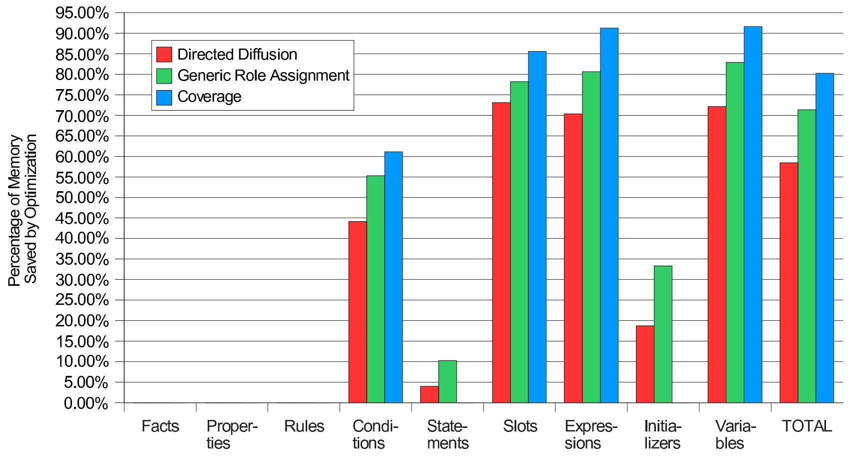

Figure 2. Percentage of memory saved through bytecode optimization for each type of programming construct.

3. Evaluating Language Constructs: Directed Diffusion

After formally introducing the concepts of RDL and describing the optimization of the resulting bytecode, we will now evaluate how far the low-level language primitives map to the requirements specific to the WSN domain. For this purpose, we will analyze in detail an excerpt of the rule-based implementation of the well-known directed diffusion communication paradigm [5].

3.1 Language Constructs in Detail

Listing 1.1 shows an excerpt of the directed diffusion ruleset, specifically the two major rules that deal with gradient management: Upon reception of an interest message and depending on whether a related gradient is already present, `addGradient` either constructs a new gradient or `reinforceGradient` reinforces a previously existing gradient.

In line 3, a fact is declared with its properties taking the role of ruleset-wide constant values in this particular case. In RDL, facts like this one are the main programming abstraction for storing data, both constant and variable, with limited scope or as part of the public interface of the ruleset. In line 4, a named slot is declared in order to allow for direct access to a property of a specific fact. In this special case however, the property was not declared manually but is rather automatically generated by the rule engine at runtime. This mechanism is used to export and control access to information internal to the rule engine, e.g. own-

Listing 1.1. Excerpt of the directed diffusion ruleset to setup and reinforce gradients.

```
 1  ruleset DirectedDiffusion
 2
 3  fact system [broadcast = 0, tx-power = 255]
 4  slot systemID = {system owner}
 5  slot systemBroadcast = {system broadcast}
 6  slot systemTxPower = {system tx-power}
 7
 8  rule addGradient 100
 9  <- exists {interest}
10  -> define gradient [sink = {interest sink}, type = {interest type},
        interval = {interest interval}]
11  -> set {interest sink} = systemID
12  -> send systemBroadcast systemTxPower {interest}
13  -> retract {interest}
14
15  rule reinforceGradient 120
16  <- exists {interest}
17  <- exists {gradient
18    <- eval ({this sink} == {interest sink})
19    <- eval ({this type} == {interest type})
20    <- eval ({this interval} > {interest interval})
21  }
22  -> set {gradient weight
23    <- eval ({this sink} == {interest sink})
24    <- eval ({this type} == {interest type})
25  } = {interest interval}
26  -> retract {interest}
```

ership of facts. In line 12, an interest message, as represented by an interest fact, is sent to all sensor nodes in radio range. Two named slots are used to easily access the information about broadcast address and transmit power, another slot defines which facts are to be enqueued for sending. As a slot can match multiple facts, it is straightforward to send several related facts with just one `send` statement, thus allowing them to be aggregated into the same packet for energy conservation. Finally in lines 17-21, the condition is shown that states that gradients are only to be reinforced if the new interval is smaller than the previous interval. This condition illustrates how elaborate slots can be used by the developer to control very precisely when a rule should fire. The firing decision is made based on the facts present in the fact repository at that time, thereby explicitly supporting the implementation of data-driven algorithms. For readability, complex slots can be declared as named slots and then reused wherever applicable.

4. Evaluating Programming Patterns

We will now concentrate on the big picture of rule-based programming rather than on specific features of RDL or the FACTS runtime. Rule-based programs differ significantly from the traditionally used imperative programs. There is no explicit flow of control and ,thus, there are no loops or conditionals. Instead of this single-chip focussed approach, we propose a data-driven and event-centric programming model as a superior programming model for WSNs. The conditions of the rules define which data items or events, both represented as facts, are to be processed, and the priorities of rules define the order in which rules react to events, possibly consuming them and thus preventing further processing.

Based on the experience gained during the aforementioned implementations of directed diffusion, GRA, and coverage, we have identified the following patterns in rule-based programming:

Event-Centric Divide & Conquer: The processing of external events is central to the area of application of WSNs. The more semantical information a single event conveys, the more relevant it is to the user. Further, aggregated events conserve energy by avoiding network traffic. When developing applications, complex events need to be broken down iteratively into simpler subevents. We refer to this process of mapping application-specific events to the actual sensor as *event-centric divide & conquer*.

In FACTS, this programming pattern maps to a hierarchy of rules: At the bottom of the hierarchy, rules fire as a result of new sensor readings and generate facts if the reading satisfies application-dependant specifications. As a second step and further up the hierarchy, a rule fires as soon as multiple of the previously generated facts are present, possibly retracting these facts and defining a new fact that represents the aggregated event.

Chaining of Filters: When information is transmitted across a WSN, each node needs to decide whether the piece of information contained in a particular packet is relevant to its current task. Dropping unnecessary information as early as possible saves both processing time and memory on each sensor node. A newly received packet is thus subjected to a series of checks, a programming pattern which we call *chaining of filters*.

FACTS supports these chain of filters by evaluating rules according to their priority: Several rules are written to fire when the same fact, which in this case may represent information that was sent over the network, is present. Each rule decides whether the fact satisfies one particular condition, e.g. whether or not it has been received previously, and retracts the fact if the condition matches. Rules that cover the most common cases are given a higher priority in order to avoid unnecessary evaluation of other rules.

State Machines: Finite state machines (FSMs) are a programming concept that is used frequently in the area of network protocols. Further, FSMs map nicely to the problem of role selection for individual sensor nodes.

Implementing FSMs in FACTS is straightforward: The current state of the FSM is stored in one or more properties of one specific *state fact*. Several *state transition rules* evaluate this fact and each one fires if the fact holds a different state. As part of the statements of the firing rule, a new state is stored in the fact, thus causing the next rule to fire. In order to model I/O, the state transition rules may additionally evaluate whether an external input fact is present, or define external output facts when firing.

Producers & Consumers: In data-driven WSN applications, there commonly is a large set of sensor nodes that gather environmental data and only a small set of sensor nodes that interface with external applications. Concentrating on the data items, they are produced at the first type of nodes and consumed by the second type of nodes.

Additionally to the automatic properties of facts provided by the runtime environment, e.g. ownership and modification time, we found it useful to add properties to facts that describe exactly under which application-specific circumstances they were produced and, if in transit through the network, how to locate their consumer. Producer information may contain node ID and sensor type, while consumer information also includes identifying the application in question.

The programming patterns described above are the result of the experience gained while implementing only three rule-based applications. It is noteworthy that especially the knowledge about *chain of filters* and *producers & consumers* helped us speed up the development process of the third application. We expect more patterns to emerge once further applications have been implemented.

5. Related Work

A reasonable level of abstraction from the underlying hardware and networking issues of wireless sensor networks has to be provided to allow for mature deployments in the future. This level of abstraction is versatile, ranging from complete middleware implementations [8], library functionalities [2], support for scoping and collaborative tasks [10] or dedicated runtime environments [11] up to language support for event-centric programming or so called macroprogramming primitives. All of these flavors aim at simplifying application development for a potentially large, distributed network of embedded sensors.

The work that has the most resemblance to ours is the Token Machine Language (TML) as described in [9]. Just as in FACTS, the authors argue that a 'unified abstraction for communication, execution and network state management' will be of great benefit for the developer. Hence, they provide tokens as a means to disseminate information in a network, and corresponding token handlers which will automatically be scheduled for execution upon reception. While in TML a token handler is conceptually a part of a token, and while it therefore provides a one-to-one mapping of tokens to handlers, our rule-based approach allows for more degrees of freedom with its many-to-many concept. A fact can trigger an arbitrary set of rules, or a rule may fire only after the system reached a certain state, expressed as a combination of several facts. Memory management in TML is divided into two sections: a fixed size shared memory is reserved for interaction between tokens while token objects (being the persistent portion of a token) will be held in a token store. In contrast, any data has to be represented as a fact in our model, making the fact repository the only means to allocate memory. FACTS is a runtime environment for rules, thus it sacrifices computational performance for a type-safe, sandboxed execution of rules optimized bytecode. Rather then providing such a runtime environment, TML is a compiled intermediate language that other high-level languages may target, and thus make use of its communication and execution semantics.

Kasten et al. address a major drawback of event-driven programming in [6]: They observe that event-centric implementations suffer from having to pass extra information between the code fragments that react to events. Specifically, there are two distinct cases referred to as manual stack management and manual control flow. Manual stack management is the use of global variables instead of automatic local variables, which are not available for sharing between event handlers. Manual control flow is code that implements different behavior of event handlers depending on a global state. This code may be duplicated in several event handlers, which results in more difficult maintenance. To solve this problem, the Object State Model (OSM) is proposed. It enhances Harel's statecharts with the option to store typed data as part of each state in so-called state variables. The scope of state variables includes, apart from obviously the state itself, all other states that are recursively embedded into it, as well as entry and exit functions of the state. As event handlers run to completion, there can be no race conditions in accessing state variables. In order to allow for this concept to be used in applications, the OSM specification language is proposed. FACTS in contrast does not allow a programmer to specify local variables at all. State is captured as a fact in the fact repository that is globally accessable by all rules. On the other hand, this will prevent a node from suffering from memory leak, since memory access is completely supervised by the FACTS runtime, and thus memory consumption on the stack can be determined at compile time. Control flow in FACTS can be easily archived with filtering strategies as described in section 1.4.

Macroprogramming, which evolves around the idea of tasking a complete network while writing a single, global program, has been explored quite early in the sensor network domain with systems like TinyDB [7] and Cougar [1]. Instead of providing such a complete, integrated infrastructure based on query processing, Kairos [4] aims at achieving a leaner support for macroprogramming. Basically, the authors extend a language with three Kairos constructs, enabling a simplified node allocation, an easy access to the one-hop neighborhood of a node and support for loosely synchronized remote data access. These extension will be annotated by a preprocessor and managed during execution on the nodes by the Kairos runtime. While Kairos optimizes for global behavior, the FACTS programming primitives have been designed for local decisions on the nodes. Global interaction may be incorporated in our model by defining a set of rules stating the desired behavior.

6. Conclusion

Rule-based programming offers a new perspective on application and middleware development for WSNs. As can be derived from the examples discussed in this paper, it provides a very concise means of leveraging the data-centric nature of this particular application domain. It allows the developer to focus on the event-driven nature of distributed algorihtms running on a highly embedded platform, rather than having to cope with machine-specific issues such as flow-control, memory management or error recovery. By introducing rule-oriented patterns, we intend to make this programming paradigm more accessible to the developer communinity.

In the future, we plan to investigate ruleset interaction with the goal to support component-based software development and optimize the runtime performance of the FACTS rule engine.

References

[1] P. Bonnet, J. E. Gehrke, and P. Seshadri. Querying the Physical World. In IEEE Personal Communications, 7(5):10-15, 2000.

[2] D. Chu, K. Lin, A. Linares, Giang Nguyen and J. Hellerstein. sdlib: A Sensor Network Data and Communications Library for Rapid and Robust Application Development. In Proceedings of the Fifth International Conference on Information Processing in Sensor Networks (IPSN/SPOTS'06), 2006.

[3] C. Frank and K. Römer. Algorithms for Generic Role Assignment in Wireless Sensor Networks, ACM International Conference on Embedded Networked Sensor Systems (Sensys) 2005, San Diego, USA, November 2005.

[4] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming Wireless Sensor Networks Using Kairos. In: Proceedings of the International Conference on Distributed Computing in Sensor Systems (DCOSS 05), LNCS 3560, Springer, 2005, pp. 126-140.

[5] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCOM '00), Boston, USA, August 2000.

[6] O. Kasten, K. Römer. Beyond Event Handlers: Programming Wireless Sensors with Attributed State Machines, IEEE/ACM International Conference on Information Processing in Sensor Networks (IPSN) 2005, pp. 45-52, Los Angeles, USA, April 2005.

[7] S. R. Madden, M. J. Franklin, J. M. Hellerstein and W. Hong. TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks. In OSDI 2002, Boston, USA, December 2002.

[8] T. Liu and M. Martonosi. Impala: A Middleware System for Managing Autonomic Parallel Sensor Systems. In ACM SIG-PLAN, San Diego, USA, June 2003.

[9] R. Newton, Avind and M. Welsh. Building up to Macroprogramming: An Intermediate Language for Sensor Networks. In Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05), April 2005.

[10] L. Mottola and G. Picco: Programming Wireless Sensor Networks with Logical Neighborhoods. In Proceedings of the 1st International Conference on Integrated Internet Ad hoc and Sensor Networks (INTERSENSE 2006), Nice (France), May 29-31, 2006.(To appear)

[11] P. Lewis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In ASPLOS-X, San Jose, USA, October 2002.