Freie Universität Berlin

Department of Mathematics and Computer Science
Institute of Computer Science

Master's Thesis

A Rule-Based Middleware Architecture for Wireless Sensor Networks

Georg Wittenburg – wittenbu@inf.fu-berlin.de

November 1, 2005

Examiner: Prof. Dr.-Ing. Jochen Schiller
Tutor: Dipl.-Inf. Kirsten Terfloth

Abstract

Wireless Sensor Networks (WSN) are an emerging technology with applications in the fields of environmental monitoring, facility management, and high-precision data gathering in general. They are deployed in the form of a large set of inexpensive sensing units that communicate over an ad-hoc wireless network. Developing applications for WSNs is challenging due to very limited resources available on each sensor node and the distributed nature of the algorithms used.

In order to ease the development effort, we propose the FACTS middleware architecture for wireless sensor networks. Built around a rule-based programming paradigm, FACTS supports event-driven and data-centric applications to be written in the ruleset definition language. Components of the middleware are implemented in rulesets, compiled into bytecode and interpreted on the sensor nodes. We illustrate the capabilities of the FACTS middleware architecture by simulating it using the `ns-2` network simulator and the ScatterWeb WSN platform.

Acknowledgments

First and foremost, I would like to thank my parents and grandparents for supporting me during my studies, providing help and advise from the humble beginnings up to and including the writing of this thesis.

Many thanks to Kirsten Terfloth, who came up with this inspiring idea in the first place, had the background, skill and nerves to hammer out all the details, and still made working on this project a thoroughly enjoyable experience.

Special thanks to Fabian Stehn for his insight, his subtle motivation now and then, and for generally keeping the place from falling apart.

Contents

Chapter 1

Introduction

One of the most fundamental observations in the computing industry is one made by Intel co-founder Gordon E. Moore in 1965 – later to become known as *Moore's Law* [Moo65] – which states:

> *"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year* [...]. *Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years."*

From this observation about how the number of transistors per chip that can be manufactured in the most cost-efficient way (given the current state-of-the-art production technology) follows the better known statement that *"the number of transistors on a chip doubles about every two years"* [int05, Sto03].

In the past decades, chip manufactures have been able to keep up with this prediction and steadily integrated more features into a single chip. At the same time the physical size of a chip has seen but little change, increasing only at a rate of about 7% per year since 1970. Amongst other things, this development has led to chips optimized for processing power as found in nowadays personal computers.

Moore's Law however also allows for a different optimization: Instead of emphasizing processing power, one can as easily optimize for low power consumption and chip size. The industry's shift towards power saving already has become visible in the latest generation of processors specifically targeted towards laptops and other mobile devices. Still, the physical size of the chip is not of major importance in this sector as devices are built to be operated directly by humans and thus have a given minimum size.

But what if one was to aggressively optimize chips for physical size and low power consumption, even at the expense of processing speed and features? How small can devices be and what capabilities can they retain? What are possible areas of application for not a

single one, but for a set of many of these small devices? The research into so-called *"wireless sensor networks"* provides a vision that gives answers to these questions.

1.1 Wireless Sensor Networks

The concept of Wireless Sensor Networks (WSN) started to appear at the end of the 1990s with first publications about Wireless Integrated Networked Sensors (WINS) [ADL+98] and Smart Dust [WLLP01]. The vision is to construct a network of very small and inexpensive devices that are connected via radio and operate unattended. The important shift in paradigm is that the individual device is no longer of crucial importance, but that the devices cooperatively – or rather the network as a whole – perform a specific task. Saying that *"the whole is more than the sum of its parts"* is especially true for wireless sensor networks.

1.1.1 Definition and Challenges

A wireless sensor network consists of a potentially very large set of individual *sensor nodes*. Each of these nodes has very limited resources, both in terms of processing speed and memory. Further, the power available to each sensor node is also limited to the point at which some approaches consider harvesting of energy from the surrounding environment to be a worth while alternative. Depending on the task of the sensor network, each node may be equipped with a different set of sensors to monitor the environment, e.g. sensors to detect temperature, motion, light, etc. Scenarios for both homogeneous wireless sensor networks, in which all nodes have the same hardware setup, and heterogeneous wireless sensor networks, in which some nodes are equipped with specialized hardware to perform their task, are conceivable.

The crucial component of a sensor node is however the radio transceiver which allows each individual node to exchange data with the rest of the network. As all other components of a sensor node, it is limited in its capabilities, hence each sensor node will only be able to reach a potentially very small fraction of the network with a directl radio transmission. To make things even more challenging, communication via radio is the most energy-demanding single action a sensor node can take and should therefore be avoided whenever possible.

Wireless sensor nodes only have a very limited user interface, if any. This fact, and even more importantly the great number of nodes deployed as part of a sensor network, makes it infeasible to configure or service them after deployment. There are even scenarios in which the exact location of each sensor node in the field is not known. The nodes of a wireless sensor network therefore need to be able to automatically configure themselves after deployment using the limited knowledge they can gather from other nodes in their vicinity. The configuration process and with it the structure of the resulting network and

the different roles that nodes perform within this structure depend on the task at hand and will differ from one deployment to another.

Unless equipped with components to harvest energy from the surrounding environment, such as photovoltaic cells, the lifetime of each sensor node and thus of the entire network will be limited. Power consumption will vary between nodes depending on network structure and the roles that sensor nodes perform during the operation of the network. However, the total lifetime of the network can be extended by intelligently restructuring the network or reassigning roles of nodes, provided that we are dealing with a densely deployed network.

Summing things up, the key restrictions of wireless sensor networks are limited computational power and energy resources. This leads to key challenges which include programming the sensor network in such a way as to make the best possible use of the scarce resources, perform with an adequate level of service under difficult conditions, increase the lifetime of the sensor network, and complete the application-specific task.

1.1.2 Motivation and Applications

While the research interest might indeed have been sparked through the possibilities offered by Moore's Law, there are several other motivations behind the research into wireless sensor networks.

Research in this area follows the general trend towards ubiquitous (or pervasive) computing. The vision is for embedded computers to become part of the environment, sensing their surroundings and intelligently react to changes. This will open the door to new ways of human-computer-interaction, such as intelligent buildings that individually adapt to persons as they move through them.

Furthermore, wireless sensor networks have applications in high-fidelity data gathering. Scenarios may be static, e.g. environmental or habitat monitoring as used in the fields of geology or biology, as well as dynamic, e.g. in situations of disaster recovery. The amount of sensor nodes deployed may vary greatly depending on the task of the wireless sensor network: If the task involves monitoring a continuous physical region, the number of sensor nodes can be potentially very large, ranging in the hundreds or even thousands. On the other hand, if sensor nodes are used to monitor discrete objects in the physical world, such as the machinery in an industrial plant, the total size of the sensor network can be rather small due to the one-to-one relationship between sensor nodes and objects of interest.

Finally, it must be noted that wireless sensor networks also do have military applications, most notably the gathering of battlefield intelligence to be used directly by soldiers in the field. It is no surprise that research is funded in part by the Defense Advanced Research Projects Agency (DARPA), an organization of the Department of Defense of the United

State of America. This raises moral issues which we are aware of but will not discuss in this thesis.

1.1.3 Platforms and Deployments

By now there is a great variety of different hardware platforms for wireless sensor networks and introducing all of them is beyond the scope of this chapter. Instead, we will describe two representative examples before turning to the ScatterWeb platform used in our own work in Section 1.2:

Smart Dust: While already finished in 2001, the Smart Dust project provided much of the vision that drives research in this area even today. The goal of Smart Dust was to develop a platform for sensing and communication with the physical size of merely one cubic millimeter. Figure 1.1 shows a Smart Dust mote as developed in mid 1999. It has a volume of about 100 cubic millimeters and comprises an optical transmitter array, an optical receiver, a charge pump, and a tiny digital controller, all mounted on top of a hearing aid battery.

Berkeley Motes: Nowadays, these are the most widely used hardware platform for implementing wireless sensor networks. Figure 1.2 shows a MICA2DOT mote, with a diameter of 25 millimeters. It is based on the Atmel ATmega128L low-power microcontroller, includes a multi-channel radio transceiver, and runs the TinyOS operating systems. Furthermore, it comes with an on-board temperature sensor, a battery monitor, and a LED, as well as expansion pins for analog, digital, and serial I/O. The mote supports routing packets and may be reprogrammed over the air.

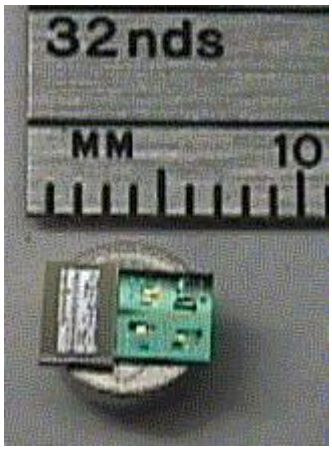

Figure 1.1: Smart Dust mote. *(Image taken from [Pis])*

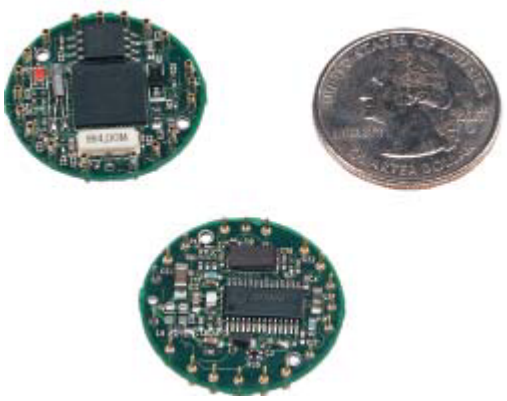

Figure 1.2: MICA2DOT mote. *(Image taken from [mic])*

All of these platforms are beginning to undertake real-world testing as part of first experimental deployments. Two examples, one using the Berkeley motes platform and the other one using the ScatterWeb platform are pointed out here for better understanding:

Great Duck Island: Starting in 2002 a wireless sensor network comprising 32 Berkeley motes was deployed on the uninhabited Great Duck Island, Maine. The goal was to develop and test a kit for non-intrusive and non-disruptive habitat monitoring, specifically sampling the microclimate in the vicinity of nesting burrows. Figure 1.3 shows two of the motes used. [SPMC04] describes the results in detail.

Heathland Experiment: Undertaken in early 2005, the goal of the Heathland Experiment was to verify lifetime predictions that until then were extrapolated based on simulation results or short deployments of sensor nodes in a lab environment. 24 ScatterWeb sensor nodes were deployed in Northern Germany and exercised by periodically building a depth-first search tree. The sensor network remained operational for two weeks. Figure 1.4 shows one of ScatterWeb sensor nodes, [TRV+05] describes the experiment in detail.

It is very likely that we will start seeing more and larger scale deployments in the near future. As both platforms presented above also have commercial offerings, there is even a chance of some of these rollouts having a business background.

Figure 1.3: Two Berkeley sensor motes deployed on Great Duck Island. *(Image taken from [gre])*

Figure 1.4: A ScatterWeb Embedded Sensor Board (ESB) node part of the Heathland Experiment. *(Image taken from [hea])*

1.1.4 Basic Services

Services required in the wireless sensor network domain differ from those already available in traditional network stacks. Consider the following examples which are unique to WSNs:

Coverage is a distributed algorithm that increases the total lifetime of a sensor network. It is applicable in scenarios in which sensor nodes gather data related to the physical location at which it is sampled and the sensor network is densely deployed. If several sensor nodes sample data from the same region, then some nodes may switch into a low power state while the region remains *covered* as far as data gathering is concerned by the neighboring nodes. Once these nodes run out of energy, the previously inactive nodes continue their task.

Clustering is the grouping of sensor nodes into logical clusters based on a shared property. Commonly, this property is the distance between nodes, as measured in the amount of hops that a network packet has to travel between them. Grouping nodes that are in close proximity with regard to network connectivity is the basis for building a hierarchical structure in an ad-hoc network, based on which more complex services such as hierarchical routing can be implemented.

In-Network Aggregation is a refinement of data-centric programming. Rather than merely collecting data from the sensor nodes and sending it to the interested parties, the data can be aggregated within the network. Performing application-specific processing of data, e.g. by calculating averages, reduces the overall data that needs to be sent through the sensor network and thus saves energy. Note that the exact operations to be performed depend on the application, i.e. the application logic is decentralized and located in part on every sensor node.

Object Tracking is a high-level application, possibly using the services provided by the items discussed above. The goal is for the sensor nodes to identify a moving object, like a specific car or animal, in the physical world and then coordinate among themselves to record its trajectory as it moves through the region of interest.

Note that most of these services require a holistic view of the sensor node, i.e. they do not fit into the layered architecture of traditional network stacks. Once again, this raises questions regarding which programming abstractions are useful in the domain of wireless sensor networks and which are not.

1.2 ScatterWeb

ScatterWeb is the platform for wireless sensor networks developed by the Computer Systems and Telematics (CST) working group at the Institute of Computer Science at Freie Universität Berlin. It is built based on the Texas Instruments MSP430 microcontroller and the TR1001 radio transceiver which operates at 868 MHz with data rates up to 19.2 kbit/s. The ScatterWeb platform comprises the Embedded Sensor Board (ESB), Embedded Chip Radio (ECR), the Embedded Web Server (EWS), and several tools and accessories.

A picture of the Embedded Sensor Board is shown in Figure 1.5. Running on three AA batteries, it includes 2 KB RAM and 8 KB EEPROM. For data gathering, it features sensors to sample temperature, luminosity, noise, and vibration. For non-radio I/O it includes an IR sender and receiver as well as a serial port. Interaction with humans is possible by the means of two buttons, one of which is freely programmable, three LEDs and a beeper.

ScatterWeb, being aimed primarily at educational institutions, has an active development community. Available software includes among many other items a gateway to GSM phone networks, various routing algorithms, clustering, time synchronization, and two virtual machines [WLT, Pie05].

The homepage of the ScatterWeb project is located at `http://www.inf.fu-berlin.de/inst/ag-tech/scatterweb_net/`. ScatterWeb components are commercially available at `http://www.scatterweb.net/`.

Figure 1.5: ScatterWeb Embedded Sensor Board (ESB). *(Image taken from [sca])*

1.3 Problem Statement and Proposed Solution

Given the wide-spread use of traditional networking technologies, it may be surprising how challenging even simple tasks become in the realm of wireless sensor networks. Nowadays an application-level developer expects certain services to be provided by the operating system or by network protocols. However, due to the resource constraints of sensor nodes neither of these can be taken for guaranteed.

Problems arise mainly from developing applications comparatively close to the hardware on one side, and the need for distributed algorithms in wireless sensor networks on the other side. Implementing a middleware as an abstraction layer is widely accepted as a solution to this problem. However, as the middleware uses some of the scarce resources of the sensor nodes, it must be carefully deliberated whether a middleware does add value from task to task. Of course, the value of a middleware increases with the features it comprises, while using as few resources as possible. The key to success is however to provide the application-level developer with a programming abstraction that empowers him to directly formulate the crucial parts of his application in a way that naturally maps to the WSN platform.

In this thesis, we propose the FACTS middleware architecture for wireless sensor networks as a solution to the problem described above. FACTS is a rule-based architecture that captures the data-centric and event-driven semantics inherent to wireless sensor networks. It includes a programming language to easily specify how to process data items in response to events occurring on the system, thus precisely capturing the desirable properties of programming in an embedded WSN environment.

Inspired by concepts from the domain of expert systems, our main programming abstractions are *facts* to encapsulate any kind of data, *rules* to asynchronously detect events and modify facts in response, and *functions* to directly harness to power of the hardware platform. Facts are stored locally in a *fact repository* on each sensor node, but may also be transmitted from one node to another. To support modularity, rules are organized in components called *rulesets*. Each of these concepts will be described in full detail in Chapter 4.

As illustrated in Figure 1.6, the FACTS architecture comprises a compiler and two runtime environments: The FACTS-rc ruleset compiler translates applications implemented in the form of rulesets to either bytecode that can be interpreted by the FACTS-re runtime environment running on the ScatterWeb platform, or to FACTS-hs definitions to run on a simulation of the FACTS middleware implemented in Haskell. Additionally, the FACTS-re runtime environment may also use the ScatterWeb on `ns-2` wrapper to execute bytecode as part of a simulation on top of `ns-2`. The details of these components will be describes in detail in Chapters 5 and 6.

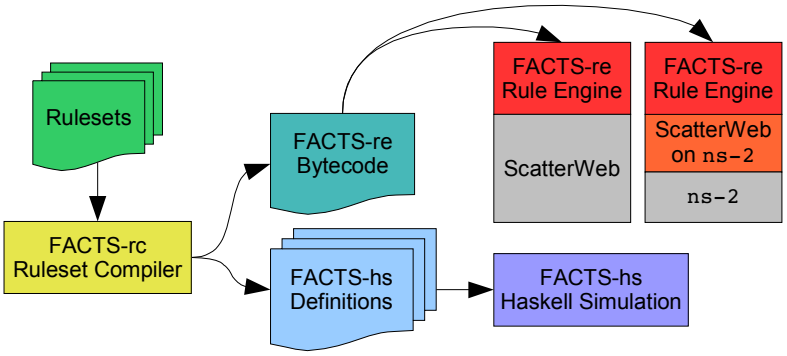

Figure 1.6: Overview of the FACTS middleware architecture for wireless sensor networks.

The next chapters of this thesis are organized as follows: In Chapter 2 we discuss related work and compare our proposed solution to other middleware approaches. Tools and procedures used while working on this project are introduced in Chapter 3. With Chapter 4 we begin the main body of our work, and describe in detail the underlying concepts of the FACTS middleware architecture as well as syntax and semantics of the ruleset definition language. On the more practical side, Chapter 5 explains the details of the implementations of the components that FACTS comprises: the FACTS-rc ruleset compiler, the FACTS-hs implementation of FACTS in Haskell, and the FACTS-re implementation of FACTS on the ScatterWeb platform. In Chapter 6 we deviate slightly from our main topic and propose a way in which ScatterWeb sensor nodes can be simulated using the `ns-2` network simulator, thereby easing our development work on the implementation of FACTS. In order to underline the relevance of our proposal, Chapter 7 has an extended use case that illustrates how other middleware concepts can be implemented in FACTS. Finally, Chapter 8 discusses future directions of research and concludes.

Additional information regarding the FACTS middleware architecture for wireless sensor networks is available on the project homepage located at `http://page.mi.fu-berlin.de/~wittenbu/uni/facts/`.

Chapter 2

Related Work

Wireless sensor networks are the focus of attention of several research groups, including (but not limited to) the Wireless Embedded Systems (WEBS) Project of the Computer Science Division at the University of California at Berkeley, the Center for Embedded Networked Sensing (CENS) at the University of California at Los Angeles, the Distributed Systems Group at the Eidgenössische Technische Hochschule Zürich, and the Computer and Network Architectures Lab at the Swedish Institute of Computer Science, among others.

In this chapter we will summarize the current state of research and discuss aspects relevant to our own work. In Sections 2.1 and 2.2 we introduce common programming tools in the area of wireless sensor networks and motivate the need for cross-layer networking. Concentrating on middleware architectures, we summarize concepts and challenges in Section 2.3 and discuss specific implementations in Section 2.4. In Section 2.5, we cover approaches towards simulation of sensor networks.

2.1 Programming Tools and Abstractions

While certainly complex, the publications discussed in this section are at a conceptually lower level, either because of dealing with direct hardware interaction or tool development. As such, we consider them tools for building a middleware architecture.

2.1.1 TinyOS

TinyOS is the most widely used operating system for the nodes of wireless sensor networks. Developed at the University of California at Berkeley and described by Hill et al. in [HSW+00], it is built around the concept of components that consist of a set of command handlers, a set of event handlers, an encapsulated fixed-size memory frame, and a bundle of threads of execution. Declarations of command and event handlers are the public interface of a component, while the frame is used to hold internal state, and threads perform computation. Threads are run by a simple FIFO scheduler and may be preempted by events,

but are otherwise isolated from each other and run to completion. For deployments a configuration of several interconnected components is created. Low-level components interact with the actual hardware, while high-level components encapsulate the application logic. The top-down flow of control is handled by commands, the bottom-up notification system is handled by events.

Applications written for TinyOS have a very small memory footprint, with the examples given in the publications ranging in the order of magnitude of a few hundred bytes. By design, an application is deployed only with those components that it depends on as stated in its configuration, hence not wasting any memory. Multiple simultaneous flows of control through the stack of components are supported, however switching between threads does incur some overhead.

In contract, the FACTS middleware architecture as proposed in this thesis is only marginally concerned with implementing access to hardware. We rather use the existing firmware of the ScatterWeb sensor nodes to this end. Similar to TinyOS, we support event-centric programming with the concepts of facts and rules, but actually take it one conceptual step further by having a first class abstraction for events in the form of facts. Similar to components in TinyOS, the FACTS rulesets encapsulate functionality and given their dependency declarations can be arranged into a similar tree-like configuration. The key difference is that in our approach the data is separated from the processing components, and we can thus support cross-layer optimization while TinyOS suffers from its excessive modularity.

2.1.2 nesC

The nesC programming language as proposed by Gay, Levis and von Behren in [GLvB+03] is the language of choice for implementing TinyOS components. We discuss it separately from TinyOS because we intend to focus on the language and tool chain features rather than the runtime environment.

nesC is an extension of the C programming language that adds support for structuring applications into components. At the same time, the extension reduces the expressive power of C in order to improve safety: For instance, the language does not support dynamic allocation of memory. As program size can safely expected to be limited, configurations are compiled as a whole rather than each component separately. As a results, the entire call graph of an application is known at compile time, thus allowing for additional program analysis and optimization to improve safety and runtime performance. The compiler has support for checking the code for data races between multiple threads of execution. Data races can be resolved by marking critical sections with the `atomic` keyword.

The imperative programming paradigm as followed by nesC is quite different from the

rule-based paradigm that is the foundation of the FACTS ruleset definition language. There are advantages and disadvantages to this: The FACTS ruleset definition language does not need to care about potentially insecure memory access or concurrency issues, because the rule engine that interprets the rules on the sensor nodes takes care of this. On the other hand, nesC benefits from its similarity to C because it is easier for new programmers to develop applications, while our ruleset definition language may be unfamiliar. As far as performance is concerned, it is still too early to tell: The ruleset definition language allows for aggressive optimizations to be performed, but at the moment it is unknown whether this can make up for the overhead incurred by rules being interpreted rather than running natively.

2.1.3 Attributed State Machines

Kasten and Römer address one major drawback of event-driven programming in [KR05]. They observe that while superior from the theoretical point of view, 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, each of which is implemented its own scope. 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[1] 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.

In our work we have identified similar problems to those described above: State is stored in facts and if different rules should fire depending on the current value of the state, the conditions of these rules need to be adapted for each possible value of the state fact. This may result in duplicated code, particularly in duplicated conditions. The solution to this may include introducing a hierarchical structure in either rules or conditions, however we have not started looking into these kinds of refinements at present.

[1] Statecharts is a model for hierarchical finite state machine (FSM) with support for concurrency. [Har87]

2.1.4 Protothreads

There is no reason why innovative concepts need to be very complex. In [DSV05], Dunkels, Schmidt and Voigt propose a creative way to use the standard C `while`-loop nested inside a `switch` statement. Their programming construct, which can be encapsulated nicely in `defines`, recreates semantics similar to those of real threads – hence the name protothread – but with a memory overhead of only one unsigned integer. Compared to event-driven programming, protothreads do not require finite state machines to be designed. Compared to traditional threads, a protothread cannot span over several C functions. It may call other C functions, but not block inside these. On systems with very limited memory, these disadvantages are however compensated by the fact that a protothread does not require its own control stack.

Protothreads themselves – being an optimization for sequential programming in C-like languages – are not directly related to our work. Still, we found the semantics quite intuitive and are contemplating whether similar constructs would be a worthwhile addition to our ruleset definition language.

2.2 Cross-layer Networking

Networking subsystems are commonly implemented as a layered architecture, the so-called network stack. Each layer performs a well defined or even standardized task, implementing a predefined interface to be used by the upper layer. This strict layering makes implementations of specific layers interchangeable and improves robustness. Until recently, this level of modularity and data hiding was regarded as a Good Thing. In the domain of wireless sensor networks however, it has been observed that more resource efficient solutions can be achieved by sharing information between layers. The technique of sharing information between layers of the network stack to boost performance has been discussed outside of the domain of wireless sensor networks under the label *"cross-layer networking"*.

Safwat, Hassenein and Mouftah were among the first to evaluate cross-layer designs for WSNs in [SHM03]. They propose two cross-layer algorithms: Energy-Constrained Path Selection (ECPS) and Energy-Efficient Load Assignment (E2LA). ECPS employs a reward-based scheme to maximize the probability of sending a packet to its destination in at most a given number of transmissions. E2LA distributes the load of routing packets over several routes, again using a reward-based scheme for route assignment. Various reward schemes using cross-layer information are presented and evaluated.

In [KKT04], Kozat, Koutsopoulos and Tassiulas tackle the problem of minimizing total transmit power while meeting predefined quality of service (QoS) parameters, such as min-

imum bandwidth or maximum bit error rate. They model network links to use slots of predefined size in packet frames, and then allocate slots based on bandwidth requirements and schedule frames to avoid collisions. Several heuristics to jointly solve these problems are discussed. Unfortunately, as the authors state at the end of their work, their algorithms need to run on a central agent with global knowledge. This constraint makes it hard if not impossible to apply their results to ad-hoc networks with only localized knowledge. Still, this paper illustrates adequately how cross-layering helps to find solutions to conflicting goals, like conservation of energy vs. quality of service.

A more holistic approach to the difficulties of power control in wireless ad-hoc networks is taken by Kawadia and Kumar in [KK04]. They begin with a very detailed list of how different aspects in the operation of a wireless network are affected by power control. For example, they point out that adjusting transmit power may cause bi-directional network links to become uni-directional or vice-versa, thereby invalidating logical structures at higher levels of the network stack. Or underlining the complexity of the problem, the value for the optimal power level is stated to be affected by the current load in the network: For high network load low transmit power achieves better results because it reduces the chances of collisions at MAC level. In contrast, for low network load it proves superior to increase transmit power to reduce the number of hops required to reach the destination node. Several algorithms are presented to find the optimal power level: COMPOW adjusts the nodes to globally use the same minimal power level that keeps all nodes of the network connected. CLUSTERPOW reduces the transmit power within the same cluster of nodes and only uses higher power levels for inter-cluster communication. The drawback of both algorithms is that they are rather computationally intensive, possibly being targeted at palmtop sized devices rather than at embedded sensor nodes.

The same authors discuss implications on a larger scale in [KK05]. The fundamental argument is that cross-layer designs have only short term benefits while an architecturally sound design is superior in the long run. As architectural layers are broken, it becomes impossible to optimize parts of the system separately, and hence optimization of a cross-layered design is far more difficult. In order to create a sound cross-layer architecture, great care must be taken to keep the interdependencies structured, well-understood and under control.

Based on these observations, we consider it crucial for a middleware architecture to support cross-layer designs. The opportunities for making better use of the limited resources available in wireless sensor networks cannot be left unexplored. At the same time we take the points raised in [KK05] serious: In the FACTS middleware architecture we support cross-layering by leaving the fact repository unstructured with each fact being available to all part of the system, as long as it is exported by its corresponding ruleset. As for the rules,

we have already a very powerful structuring mechanism in place: By adjusting the priority of a rule (as described in Section 4.2.2) it can either fire early or late during the run of the rule engine. Firing earlier moves the rule conceptually closer the generation of the event, i.e. to a lower layer in the network stack. Formalizing priority ranges to correspond to traditional layers of the network stack will leave us with the flexibility of cross-layer design and the structure provided by rulesets at the same time.

2.3 Middleware Overview and Concepts

Römer, Kasten and Mattern summarize the requirements on middleware architectures in the domain of wireless sensor networks in [RKM02]. According to them, the main purpose of middleware is to support development, maintenance, deployment, and execution of applications or services. Key features of a middleware architecture are energy efficiency, robustness, and scalability. The middleware should also be able to connect the WSN deployed in the field to external components for control and data storage, while at the same time offering support for establishing time and location when gathering sensor data. Furthermore, they emphasize the design principles of application knowledge being available on the sensor nodes and data-centric communication between the nodes. Both of these principles help to make the best possible use of available resources. Finally, as sensor nodes operate unattended after initial deployment, the middleware must provide for or support automatic configuration and error handling.

In [TS05], Terfloth and Schiller classify existing middleware approaches into three conceptual groups: Group abstractions, such as Generic Role Assignments (which we will discuss in Section 2.4.4), organize the sensor nodes into groups that share relevant characteristics. A very common characteristic for instance is network proximity. Grouping nodes based on this metric results in clusters of nodes, which are the basis for hierarchical addressing and eventually routing. A second conceptual group consists of those middleware approaches that implement a virtual machine. Their main goals are to provide an abstraction layer from the possibly heterogeneous hardware platform of different sensor nodes and to improve robustness by running applications in a safe execution environment. One example of virtual machine abstraction is Maté (see Section 2.4.2). The third and final conceptual group comprises those middleware architectures that focus on modularity and service orientation. Modularity allows for more flexibility in the software development process, and the definition of services makes it possible to reduce the software features deployed to those that are actually required by the application, thus saving resources on the sensor nodes. The two concepts complement each other, as services are commonly implemented in the form of interchangeable modules. Each of the three conceptual groups has certain advantages

desirable for wireless sensor networks. However, existing middleware approaches tend to be specialized to merely one concept, in some cases only slightly overlapping with a second one.

In the FACTS middleware architecture we combine the benefits of the three conceptual groups as into one system. We expect the resulting middleware architecture to meet and exceed the challenges discussed above.

2.4 Existing Middleware Approaches

None of the concepts used in FACTS is completely new, most have been used with slightly different semantics and under different names in other middleware approaches. In this section we will directly compare FACTS to these approaches, discussing similarities and differences. We will illustrate how concepts and abstractions of the FACTS middleware architecture map nicely to the application domain by pointing out how different subsets of them have been used in the implementation of other projects.

2.4.1 Directed Diffusion

Proposed by Intanagonwiwat, Govindan and Estrin in [IGE00], Directed Diffusion is a data gathering framework for wireless sensor networks. Queries, or rather *interests* are structured to support request based on different types of events, total duration during which a query is to be active, the interval in which sensor readings are to be sampled, and a geographical area of interest. Queries can be injected into any sensor node and are disseminated automatically through the network. During dissemination interests leave a trail of *gradients*. If a sensor node detects an event that matches one of the interests it has received so far, it sends the data back along the gradients that were left by the interest until the data eventually reaches the original source of the interest. Note that this process only requires local knowledge on each sensor node. Gradients are reinforced when the source renews its interest and the interests are propagated along a previously used path in the network. Over time, the optimal route between the data gathering nodes and the data sink is established.

Directed Diffusion is a rather specialized architecture that concentrates on solving the problem of data gathering and potentially in-network aggregation. Complimentary services, for example hierarchical routing, are however hard to integrate. Compared to FACTS, both interests and sensor readings can be implemented as facts that are propagated automatically through the network. A FACTS ruleset that implements a similar propagation mechanism is presented in Chapter 7. Furthermore, gradients can easily be stored as facts in the local

fact repository of each node, thereby making an implementation of Direction Diffusion on top of the FACTS middleware architecture feasible.

2.4.2 Maté

Maté as introduced by Levis and Culler in [LC02] is a bytecode interpreter that runs on TinyOS. As such, it conceptually belongs to the the virtual machine category described in [TS05]. It is implemented as a TinyOS component (compare Section 2.1.1).

With the goal of reducing bytecode size, Maté is designed as a stack-based architecture, and as such does not require basic bytecode instructions to specify their operands. Some instruction go beyond the usual scope of virtual machines and make Maté particularly suitable for the wireless sensor network domain. For example, the instruction set includes a `send` instruction that implements unreliable multi-hop communication with other sensor nodes. The bytecode is broken down into fixed size capsules that may be transmitted over the network. This feature makes it possible to distribute bytecode after the wireless sensor network has been deployed, thus allowing for the capabilities of the WSN to be adjusted at runtime.

Maté is similar to the FACTS rule engine as both of them are bytecode interpreters. Our rule engine does however not implement a virtual machine because interpreting rules does not involve architectural features of traditional processing units such as a program counter or an execution stack. In fact, we try to avoid dynamic memory structures outside of the fact repository whenever possible, thus taking the constraints of the sensor nodes into account. While not implemented yet, we envision for our rulesets to be deployable at runtime, similar to Maté's bytecode capsules.

2.4.3 SWARMS

In [BFK+03], Buschmann et al. describe a middleware approach that tries to mimic the behavior observed in swarms of animals, who as a group are capable of achievements far beyond the reach of a single individual. Before this kind of emerging property can be achieved, the means of communication between the sensor nodes must be established. The authors propose a Virtual Shared Information Space (VSIS) that stores data represented in the XML format. The SWARMSware middleware coordinates access to the VSIS, retrieves, filters, and aggregates data. Higher-level services such as levels of trust into a particular data item and role assignment are also supported.

Obviously, the overhead of XML-structured data is not acceptable for a WSN middleware architecture. Except for this item, the concept of a shared information space is quite attractive. Algorithms in wireless sensor networks frequently work on data gathered from

several different sensor nodes, hence making the exchange of data transparent between nodes is a desirable feature of a middleware architecture. In FACTS we have laid the foundations for this kind of service by assigning globally unique IDs to facts and making it easy to transmit facts over the network. A ruleset implementation of a service that retrieves specific facts from the network is definitely feasible.

2.4.4 Generic Role Assignment

Römer, Frank et al. identify the assignment of roles to individual sensor nodes based on their capabilities as a recurring problem in wireless sensor networks. In [RFMB04] and [FR05] they propose Generic Role Assignment (GRA) as an abstraction from role assignment based on hard-coded conditions.

In GRA, roles may be assigned based on the capabilities and resources of the local sensor node as well as based on roles and resources of other nodes in the neighborhood. Typical roles include whether a node is switched on or off as part of a coverage algorithm, whether a node is cluster head, gateway or slave in hierarchical routing, or whether it is a source or an aggregator in a data aggregation scenario. As part of GRA, a language for role specification is proposed. At the core of the system is the distributed role assignment algorithm. As nodes may change their role in response to role changes of other nodes, care must be taken that the network as a whole eventually reaches a consistent and stable state. Further, it is desirable to avoid excessive communication by intelligently choosing a role for the local node before propagating it over the network.

Generic Role Assignment – while definitely deserving merit in its own right – is a subset of the FACTS middleware architecture. The definition of capabilities and resources map directly to our facts, the role specification can easily be expressed in the form of rules. Even implementing the role assignment algorithm in the form of a ruleset is feasible. To support these claims we discuss how Generic Role Assignment can be implemented on top of FACTS in Chapter 7.

2.4.5 TinyLIME

A different approach is taken by Curino et al. in [CGG+05]: With TinyLIME they propose a middleware architecture that is intended for cases in which highly mobile base stations interact with only a local subset of the sensor network. TinyLIME uses the idea of a Linda tuple space as abstraction for data storage and transmission, and is built on top of Linda in a Mobile Environment (LIME). The tuple space is shared and updated when sensor nodes or base stations move into close proximity of each other. The union of all these separated tuple spaces is called a *federated tuple space*, which changes over time based on

the connectivity between the sensor nodes and the base stations. A node may register patterns to be matched against the tuple space and execute a code fragment in response to an appropriate tuple appearing. This mechanism is referred to as *reactions*. TinyLIME has been successfully implemented in nesC and was simulated using TOSSIM.

The similarity between TinyLIME and FACTS is striking. Single tuples closely resemble our facts and reactions are quite similar to our rules. However, the authors do not focus on the event-like dynamics, possibly underestimating the potential of TinyLIME. In contrast, event-driven programming is a core feature of FACTS. We have already implemented fact propagation, which is equivalent to updates of the federated tuple space, as part of the use case described in Chapter 7. This mechanism is however not a built-in component of the architecture, but rather implemented as a ruleset. As such it is easy to adapt to an application-specific need for data propagation, potentially reducing network traffic. Further, our rules are interpreted on the sensor node rather than running natively as reactions in TinyLIME do. We thereby preserve the additional advantages of a virtual machine architecture.

2.5 Simulation

Simulation of wireless sensor networks is not the main topic of this thesis. However, in Chapter 6 we describe how to simulate a ScatterWeb user application on the `ns-2` network simulator with the goal of easing the development effort in the FACTS middleware architecture. In order to estimate the potential of this subproject, we discuss other approaches to simulation in this section.

2.5.1 TOSSIM

Proposed by Levis et al. in [LLWC03], TOSSIM is a simulator for wireless sensor networks that run TinyOS. Due to the component-oriented architecture of TinyOS, simulation of a sensor node is relatively straightforward: TOSSIM implements all required hardware interface components and the remaining components of the configuration run unchanged on top of these. For every node in the simulation, a separate instance of the component graph is created. Additionally, there is also a discrete event queue to simulate interaction with the hardware. Radio communication between sensor nodes is simulated at bit level with bit error rates specified for uni-directional connection in the network. The simulation also supports communicating with external programs for control, visualization, and debugging. Applications written in nesC can be compiled transparently to be executed on the the real sensor node or on TOSSIM.

The key advantage of TOSSIM is its seamless integration with TinyOS and the nesC language. As TinyOS applications are already inherently structured, it is relatively easy to replace one hardware abstraction layer, i.e. the components that interact with the hardware, with a different, simulated one. Compared to this, the ScatterWeb firmware API is quite heavy-weight and using it as abstraction layer for simulation purposes is more difficult. On the other hand, the radio simulation of TOSSIM is quite simplistic, e.g. it seems to ignore interference caused by simultaneous transmissions. By choosing `ns-2` as our simulation backend, we expect our results to be more realistic and relevant.

2.5.2 Wireless Sensor Networks on `ns-2`

This section deals with simulating wireless sensor networks on `ns-2` as proposed by Downard in [Dow04]. For an introduction to the `ns-2` network simulator itself see Section 3.3.

Downard extends the popular `ns-2` network simulator to allow for simulation of wireless sensor networks. `ns-2` already ships with built-in support for physical radio propagation models, antenna models, and wireless MAC layer protocols. The missing element for a complete simulation of wireless sensor networks is a simulation of the environment for the sensor nodes to gather their data from. To implement this feature, a class of phenomenon nodes is added to the system, to simulate real-world data sources by periodically broadcasting their phenomenon type and associated value to all sensor nodes in the region. The data is encapsulated in a newly created PHENOM packet and sent over a designated channel. Exactly which sensor nodes are affected by the phenomenon is determined by the radio propagation model included in the configuration of the phenomenon node. As all other nodes, the phenomenon node is able to move around during the simulation, thus simulating a moving data source such as an animal.

Downard's work is complementary to our ScatterWeb on `ns-2` effort. We are only concerned with simulating the sensor nodes, while he concentrates on simulating the environment that the sensor nodes operate in. However, the approach presented has certain drawbacks: Broadcasting information about phenomenons causes unnecessary simulation overhead, which is not required if the phenomenon in question is not relevant to the sensor nodes at that time. Further, using periodic updates does not allow the sensor nodes to sample the environment at a rate they are concerned about. In Section 8.2 we sketch how a more efficient and realistic subscription-based approach can be implemented.

2.5.3 OMNeT++

Mallande et al. propose their own solution to the problem of simulating wireless sensor networks in [MSK+05]. Arguing that `ns-2` is difficult to extend because of interdependencies between modules, they propose OMNeT++ as an alternative platform. Like `ns-2`, OMNeT++ is written in C++, however in the list of noteworthy features it is hard to find anything special, except possibly the quite comprehensive simulation library. When compared to `ns-2`, the OMNeT++ simulator uses less memory and simulates matching setups in less time. However, the quality of the simulation is questionable. For example, radio transmissions are simulated at the packet level, and once again interference is not taken into account.

In light of the obvious deficiencies of OMNeT++, we are quite happy to invest the time to properly understand `ns-2`. Once implemented, our simulation enjoys the benefits of running on a widely tested platform, and the simulation results are not questionable with regard to the methodology used.

2.6 Theoretical Background

For the theoretical underfittings of event-based systems we turn to Hinze and Voisard who in [HV02] describe an algebra that formally defines the exact semantics of events and event composition over time. Coming from a background in event notification systems in databases, the authors give definitions of temporal disjunction, conjunction, sequence, negation and selection. Composed events based on this terminology helped us to nail down exactly under which circumstances a rule should fire. Further, our facts – although developed independently – are surprisingly similar in semantics and structure to their notion of events, which we find encouraging.

While outside the scope of this thesis, we expect the theoretical constructs to be quite valuable when proving the correctness of algorithms implemented in the form of FACTS rulesets.

Chapter 3

Tools and Procedures

In this chapter we will quickly go over a selection of the tools that were used when implementing the components of the FACTS middleware architecture as described in Chapter 5. The goal is to introduce non-standard tools and point out caveats of well-known tools, thus making sure that the results proposed in the following chapters are easily reproducible.

Standard tools which have been used, but will not be discussed in further detail for brevity, are the GNU Compiler Collection (`gcc`) and Debugger (`gdb`), the Hugs 98 Haskell interpreter, the Flex fast lexical analyzer generator (in Lex compatibility mode), the Bison general-purpose parser generator (in Yacc compatibility mode), and, of course, LaTeX.

3.1 Project Management

Development was organized according to the Rapid Prototyping process described in [Bal98, Part II, Section 3.3.3]. The goal is to start implementing small, but still relevant aspects of the proposed solution very early in the project. These building blocks can be analyzed, refined, merged or discarded. Over time, the prototypes will begin to resemble the desired product more and more closely.

The advantages of this process include that the availability of a prototype helps in understanding all details of the problem and boosts creativity in developing solutions. Being a very hands-on approach, it encourages feedback from all stakeholders. With the appropriate tools, in our case the Haskell programming language, prototypes are comparatively easy to create. Finally, it should not be underestimated that having at least partially working prototypes available during most of the development cycle of the project reduces the risk that the project might fail. Due to these reasons, Rapid Prototyping is particularly well suited for research-oriented projects such as the one treated in this thesis.

3.2 Subversion

Subversion (with its command-line client `svn`) is a source code management system (SCM) designed as drop-in replacement of the Concurrent Versions System (CVS), but fixing some of the inherent weaknesses of CVS. Most notably, Subversion supports versioning of directories, renames, file meta-data, and even symbolic links. Additionally, commits are truly atomic, binary data is handled automatically and efficiently, and branching and tagging is done using server-side copies in constant time with a mechanism similar to hard-links. Subversion is available at http://subversion.tigris.org/.

Subversion supports multiple storage backends such as Berkeley DB and a plain filesystem-based structure. Berkeley DB is the default backend up until version 1.1.4 of Subversion, but proved to be unreliable because temporary and log files created by Berkeley DB caused permission related problems when faced with multi-user access. Unfortunately, these problems resulted reproducibly in a corrupted database and hence data loss in the source code repository. While recovery tools shipped with Subversion were unable to fix this problem, a solution was eventually found relying on Berkeley DB tools. As a consequence and following the advice on the Subversion mailing list, the repository was migrated to the filesystem-based storage backend which proved more reliable. Since version 1.2 the filesystem-based backend is the default storage backend. The full discussion on the Subversion mailing list is available at http://svn.haxx.se/users/archive-2005-05/0509.shtml.

The flexibility of Subversion when it comes to naming files greatly eases prototyping during the early stages of development, a definitive advantage over CVS. It is therefore highly recommendable for projects similar in size and scope.

3.3 ns-2 Network Simulator

The `ns-2` network simulator is the most widely used simulator in networking research. Architecturally a discrete event simulator, it allows for simulation of most modern TCP, routing and multicast network protocols over both wired and wireless network links. On the implementation side, `ns-2` is a simulation framework with support for distributed simulation written in C++, that interprets scripts written in the Tool Command Language (Tcl). Based on the simulation specified in these scripts, `ns-2` produces detailed output traces that can be examined statistically or replayed in tools such as the Network Animator (`nam`). Furthermore, it is possible to run `ns-2` with a real-time scheduler to allow for interactions between real and simulated network components. `ns-2` and related tools are available at http://www.isi.edu/nsnam/ns/.

Although extensively documented in [ns005], we found that it was rather difficult to

understand the inner workings of ns-2 up to the point at which it is possible to extend the system. Actually, reading the C++ class definitions and Tcl scripts proved far more valuable than the manual. Examining the running program with gdb is also a worthwhile technique. We were able to fix a few compilation issue in both ns-2 and nam as well as the supporting libraries tclcl and otcl. Patches have been submitted to the respective authors and were partially accepted. Further, it must be noted that ns-2 does *not* compile cleanly with the current 4.x series of the g++ compiler, but just with the older version 3.3. We have documented these findings on the homepage of the FACTS project at [Wit].

3.4 C to C++ Translator (ctocpp)

The ctocpp C to C++ Translator is a Python script that translates programs written in the C programming language into the C++ programming language. This process is not trivial, because while C is a subset of C++, it is hard to automatically generate an object-oriented structure based on a given C program. ctocpp is available at http://www.scriptet.com/ctocpp.php.

As we will explain in Section 6.2, an attempt was made to use ctocpp during the effort of porting the ScatterWeb firmware API to the ns-2 network simulator. Unfortunately, this attempt was not successful for a variety of reasons: ctocpp neither handles production-level C code, e.g. code containing compiler directives, nor resolution of header files. Recursive path resolution is slightly problematic, too. As for the generated C++ class definitions, we found it to be less than optimal, with function remaining static and no constructors or destructors being generated.

We created and submitted several patches to the author to fix some of these deficiencies. Although some of the patches were merged and the feedback was positive, we decide to abandon ctocpp in favor of a different solution.

3.5 y2l – Yacc to LaTeX

Yacc to LaTeX(y2l) is an AWK script that derives the Extended Backus-Naur Form (EBNF) from a Yacc source file. The EBNF is formatted as a LaTeXsource file and can be easily included in other documents. Generally, the most accurate description of a programming language is the one found in the definition of the compiler frontend. Hence it makes sense to automatically generate the grammar of the language based on this definition. As Yacc is a preferred way of implementing compiler frontends, the utility of the y2l script is obvious. y2l is available at http://www.alchar.org/~aedil/Projects/y2l.html.

To further ease development, we enhanced y2l to use the Lex definition file as a source

for gathering the terminal symbols of the language and fixed a few inconsistencies in the output. Patches have been submitted, but there was no feedback as of this writing.

`y2l` was used to generate the grammar of the FACTS ruleset definition language as shown in Appendix A.

Chapter 4

Concepts and Language

Given the shortcomings of existing middleware approaches for wireless sensor networks as listed in Section 2.4, this chapter will describe in full detail our proposed solution – the FACTS middleware architecture.

After a brief overview in Section 4.1, the fundamental concepts of our proposal are introduced in detail in Sections 4.2 and 4.3. Section 4.4 gives some background on the considerations that led to the exact semantics of these concepts. Subsequently, the programming language of our system – the FACTS ruleset definition language – is introduced, with Section 4.5.1 explaining the syntax and Section 4.5.2 providing some examples.

4.1 Overview

The crucial point that decides about success or failure of a middleware architecture is the question whether it provides the application-level programmer with a powerful abstraction from the underlying hardware. The main goal is to allow him to solve his programming task in a quick and efficient manner, while at the same time providing an abstract mental model that eases thinking and talking about components and algorithms of the system.

The key abstraction of our proposal is that of a *distributed expert system*. In this context, information – that is everything ranging from sensor readings to temporary variables – is represented as *facts* which are processed by *rules*. In contrast to backward-chaining inference engines, which are used to implement popular expert systems such as JESS [jes], our *rule engine*, which processes the facts according to the given rules, uses forward-chaining in order to provide event-like semantics. Furthermore, rules may also call *functions* which hook into the firmware of the sensor node and perform resource critical operations in a fast and memory efficient manner.

Using the term *"expert system"* to describe our middleware architecture is not uncontroversial. It was suggested to us that expert systems are traditionally associated with backward-chaining as opposed to event-driven programming, and, what is worse, imple-

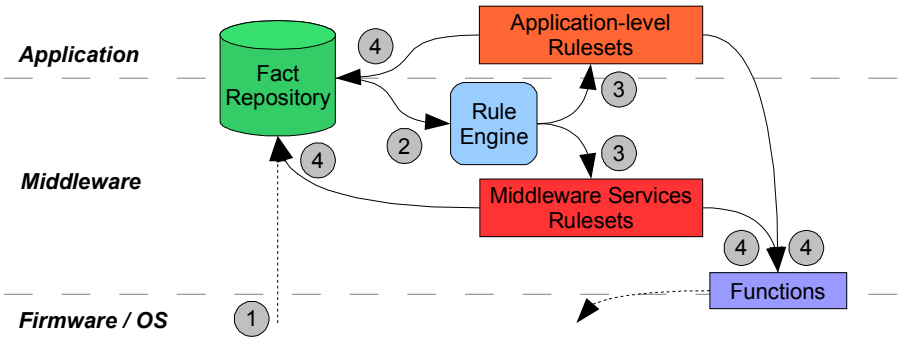

Figure 4.1: Interaction between the components of the FACTS middleware architecture.

mentations of expert systems are not particularly famous for performing well under tight resource limits. It was suggested that the analogy might confuse potential users. On the other hand, we have repeatedly found it helpful to use exactly the term *"expert system"* as a brief synopsis of what the core idea of our middleware architecture is. The term provides a well-known context in which our concepts are easier to understand when first learning about them. To address the two key concerns mentioned above, we have labeled the algorithmic core of the middleware *"rule engine"* rather than *"inference engine"*, and try hard to emphasize on low resource consumption.

Facts, rules and functions are local to each node of a wireless sensor network and each node runs its own rule engine. However, facts are also used as the key abstraction for transmitting information from one node to another, to the entire sensor network, or to a specific set of nodes. Hence, one can think of it as a node sending one of the facts from its own *fact repository* to the fact repository of another node. This node in turn knows that the original owner of the fact is the first of the two nodes and not itself. Other than that, there is no further distinction made between local facts and facts received from other nodes over the network.

While not currently implemented, it is easy to envision for rules and function to be updated after a wireless sensor network has been deployed. New rules may enhance or adapt the function (or role) that a node fulfills as part of the sensor network. New functions may allow to provide additional hooks into the underlying firmware, possibly fine tuning the sensor network for efficiency.

Figure 4.1 illustrates the interaction between the main components of the FACTS middleware architecture. Low-level events start execution by creating new facts (1) which trigger the rule engine (2) to check and possibly fire rules from available rulesets (3). The rules then may either modify the fact repository or call functions (4).

4.2 Basic Concepts

The three basic concepts of the FACTS middleware architecture are facts, rules, and functions. In the following, we introduce each of them in detail.

4.2.1 Facts

Facts are the central means of representing any kind of data in the system. They are structured as a named set of key-value-tuples. Multiple facts with the same name may be present in each local fact repository. A specific key-value-tuple is called a *property* of a fact. Property keys are unique within each fact, values are typed. Available types of values are `bool`, `int` and `string`.

Every fact has four predefined properties which are updated by the rule engine and available to the programmer as read-only values. The keys and types of these properties are:

owner (int): The network-wide unique ID of the sensor node that was the last to modify this fact by either creating it, adding a new property, or modifying an existing property.

time (int): The time at which the fact was last modified. Note that this merely is the perception of the current time of the modifying node, which may well be out of sync with the rest of the wireless sensor network. While theoretically feasible, it is currently outside the scope of the middleware architecture to provide a time synchronization service.

id (string): The network-wide unique ID of this fact. It is implemented as the dot-separated concatenation of the ID of the owning sensor node (which is unique in the wireless sensor network) and the time of last modification (which is unique on the sensor node, that sets itself as the owner upon modifying the fact).

modified (bool): This boolean flag indicates whether a fact has been modified by either another rule or some external system-generated event during or since the last run of the rule engine. As rules only fire when one of the facts referenced in their conditions has been modified, this property is useful for working on only those facts that have caused the the rule to fire and hence might require processing. The modified flag of all facts in the local fact repository is cleared every time an entire run of the rule engine has been completed.

As facts are dynamically added to the fact repository at runtime and as only facts marked as modified trigger the execution of rules, modified facts appear as events to the application-

level programmer, thus allowing for event-driven programming. Once again providing theoretical background, facts correspond in semantical meaning and structure to the concept of an *event* as described in [HV02, Section 2.2]:

> *"An* event *is the occurrence of a state transition at a certain point in time.* [...] *Events can be described as collections of (attribute, value) pairs,* [...]*"*

Furthermore, facts are also the central means of transmitting information between nodes of the wireless sensor network: A `send` statement (as introduced in the following section) in a rule transmits one or multiple facts to another node or broadcasts them to the entire network. The rule engine itself does not include higher-level services such as routing. However, these services can be implemented as part of the middleware.

4.2.2 Rules

Rules are used to express algorithms and reactions to events external to the rule engine. A rule is a named structure containing both a set of conditions and an ordered list of statements. A rule fires, i.e. the rule engine executes the statements belonging to the rule, if all the conditions become true according to the facts currently present in the fact repository. More precisely, a rule fires if all conditions are true and at least one of the facts mentioned in these conditions is tagged as modified. This ensures that the rule engine processes changes in the fact repository in an event-like manner.

To flesh out the above description with a more theoretical background, a rule fires as soon as the *temporal conjunction* (see [HV02, Section 3]) of all complex events (as defined by the conditions of the rule) since the start of the rule engine is true. Additionally, the time of one of these events (which corresponds to the modification time of a fact referenced by a condition) must be after the time at which the previous run of the rule engine started.

A rule also has a priority which schedules at which time during a run of the rule engine the conditions of the rule in question are checked against the fact repository. Based on the result, it is then decided whether the rule should fire or not. Multiple rules can have the same priority, but for these rules the system does not provide a guarantee on the order in which they are executed.

Conditions have one of two forms:

Exists: Checks whether one particular fact is present in the local fact repository at this time.

Eval: Checks whether a boolean expression is true given the data found in the local fact repository at this time.

Statements modify the fact repository or generally interact with the rest of the system. Available statements are:

Define: Adds a new fact to the local fact repository and initializes its properties, including ownership and modification time.

Retract: Removes one or more facts from the local fact repository.

Copy: Creates an exact copy of one or more facts, only updating ownership and modification time.

Send: Copies one or more facts from the local fact repository to remote fact repositories of other nodes in the wireless sensor network.

Set: Changes the value of an existing property of one or more facts in the local fact repository or adds a new property to an existing fact. Also updates ownership and modification time of affected facts.

Flush: Sets one or more facts to unmodified in order to prevent them from causing rules to fire in the the next run of the rule engine.

Call: Calls a function, possibly passing arguments based on properties of facts.

The exact syntax of these conditions and statements is given in Section 4.5.1.

4.2.3 Functions

A function is a piece of machine code that runs natively on the processor of the sensor node and interacts with the firmware. It provides an interface that can be invoked by the rule engine. As such, its main purposes are to provide hooks for the rules to interact with the hardware of a sensor node and to allow for efficient implementation of performance critical algorithms.

4.3 Derived Concepts

Complementing the basic concepts of the FACTS middleware architecture, the following concepts provide additional means of interaction and abstraction:

4.3.1 Slots

A slot is the primary mean for addressing facts and their property values in the fact repository. It is a tuple consisting of two strings, one identifying the fact and one identifying the property key, and a list of conditions that further specify which subset of the fact repository is to be addressed. To put it differently, facts that do not match the conditions of a slot are filtered out, and only the facts addressed by the slot are returned. The property key may be omitted if only the fact itself, rather than one of its properties, is of interest.

4.3.2 Rulesets

A ruleset captures the concept of a middleware component from the software engineering point of view. It is a set of rules and related facts that together provide certain services to the system. Rulesets have a well-defined interface in terms of a set of facts that trigger the execution of their rules. They encapsulate locally used rules and facts in their own namespace which is implemented as dot-separated prefix to the name of the facts or rules in question.

We plan for rulesets to explicitly provide services as identified by a well-known descriptor, or require them to be present on a local system when being installed. This will allow for the construction of a dependency graph at compile time and possibly for automatic loading of rulesets at runtime.

Experience gained by implementing several common algorithms typical for wireless sensor networks shows that a ruleset is typically in the magnitude of ten to twenty rules and a similar amount of facts. These numbers are merely rough estimates and vary depending on the complexity of the functionality that is to be implemented.

4.3.3 Globally Shared Information Space

Facts have a unique ID that is the concatenation of the ID of the owning node and the time when the fact was last modified. Only one fact can be modified by a sensor node at any given time. Using the unique ID, addressing facts on any sensor node of the wireless sensor network is possible. Taking the next logical step, it is feasible to implement services that retrieve remote facts based on their ID, thus constructing a globally shared information space.

Furthermore, a fact repository may contain various facts from different sensor nodes. For the local sensor node it acts as a knowledge base. The knowledge stored within the facts of the local fact repository comprises information about current state of the sensor node, events it has spotted during a certain time interval, or similar information of other nodes in the wireless sensor network.

4.4 Design Details and Considerations

The concepts as described above were rough sketches when the work of formalization began. Several decisions were made during the design phase of the project, which eventually led to the current semantics of the FACTS middleware architecture. This section explains important details of our design and the rationale to support them.

4.4.1 Sets of Facts

As facts are addressed by their name – which is not required to be unique – evaluating a slot against the fact repository may result in a set of multiple matching facts. Hence, a condition or a statement that takes a slot as a parameter may either process a single fact or a set of facts depending on the content of the fact repository at that time.

For a condition to return true when evaluated against a given fact repository, it is sufficient if at least one of the facts matched by a slot satisfies the constrains stated in the condition. A statement however will be executed separately for every single fact matched by the slot. In case of a statement containing multiple slots, the statement is applied sequentially to all possible combinations of the matching facts, i.e. the cross product of the respective sets of facts.

If only one specific fact is to be processed, this can either be achieved by carefully naming the facts, or by providing more specific constraints in the form of conditions as additional parameter of the slot. The following section explains this in detail.

4.4.2 Separation of Conditions and Statements

Even when belonging to the same rule, conditions and statements are clearly separate entities. Conditions only take care of firing the rule without any side effects. Statements only alter the fact repository or interact with the system.

This concept results in statements having exactly the same semantics no matter as part of which rule they are executed. However, being independent from the conditions of the rule also implies that the filtering of exactly which facts to process needs to be done for each statement separately.

4.4.3 Filtering Facts for Processing by a Statement

Apart from addressing a fact by its name, a slot is frequently required to be more specific about exactly which fact from a potentially large set of matching facts it should address. To this end, a slot can filter the set of all facts with matching names by giving further constraints about the required values of the properties of said facts. This is done in the

form of a list of conditions that can be specified as additional parameter of the slot. The result is that the slot only addresses the subset of facts that matches all conditions.

It is up to the application-level programmer to ensure that the properties of facts differ enough for him to isolate a specific single fact in case this is required by the application. The rule engine supports him by providing a unique ID for each fact in the fact repository and making it accessible via the read-only `id` property.

An alternative design would have been to implement implicit filtering of the facts based on the conditions of the rule in question. While this would allow for a simpler syntax, there are several disadvantages:

- Statements might access facts that are not referenced in the conditions. Adding conditions for filtering purposes alone would bloat the application-level code.
- It would raise questions on the order in which to process the statements: Should all statements be executed sequentially for all matching facts, or should rather each single statement be applied to each matching fact before executing the following statement? None of these two options seems intuitive enough to be acceptable by an application-level programmer.
- Statements would have different semantics depending on which rule they appear in. This would not only be confusing, but also violate the idea of strict decoupling of conditions and statements within a rule as detailed in the next section.

In light of these drawbacks, it is understandable why we decided against implicit filtering.

4.4.4 Adjusting Ownership of Modified Facts

The `owner` property of a fact stores the information on which sensor node was the last one to modify the fact in question. On read access this property remains unmodified. The goal is to keep the globally shared information space intact, e.g. ensure that sensor readings processed within the sensor network are clearly marked as such. If the original fact is to be preserved while processing, a copy needs to be made beforehand. The copy is owned by the local sensor node and can therefore be modified safely. In case only changes to facts owned by the local sensor node are intended, a filtering condition needs to be added to slots stating that only facts whose owner property matches the ID of the local node are to be processed.

An alternative solution proposes that facts should be owned by the originating sensor node exclusively. Updates would then result in changes just in case the current executing entity is also the owner of the fact and otherwise leave the fact untouched. It turned out that while allowing for the same functionality to be implemented, these semantics resulted

in bloated code: For instance, when sending facts across the network, facts can be regarded as packets, and packet properties, i.e. facts properties, need to be updated slightly for each hop they travel on the network. Having to make a copy before being able to process a packet would not only waste memory but also result in unreadable code and unnecessary processing overhead.

4.4.5 Avoiding Local Variables

Unlike JESS [jes], a reference expert system we examined for language formalization, our system does not support the notion of local variables to which a specific fact can be bound within a rule. We consider binding facts to variables at runtime to be too expensive in terms of memory usage for a sensor node. The syntax of filtering in slots is able to provide the same functionality, while lowering system overhead for memory management.

4.4.6 No `else` Keyword

We have been tempted several times to include the semantics of an `else` keyword into the ruleset definition language. At first glance, there are numerous occasions in which it seems to make sense to execute certain statements if the conditions of a rule are *not* true.

However, the semantics of an `else`-block of statements do not interact well with the logic that fires a rule in the first place: If the conditions evaluate as true, the normal statements of the rule are executed. This leaves the `else`-block executing when not all conditions evaluate as true. But then the rule would not fire in the first place. Instead, the `else`-block could be executed when a rule did *not* fire. But this would lead to most `else`-blocks being executed most of the time, even in the absence of any other changes in the fact repository which totally undermines the event-driven semantics of the FACTS middleware architecture.

So even though there are cases in which the semantics of an `else` keyword might seem desirable, on closer inspection it becomes obvious that they do not interact with the other concepts of the middleware in a way that warrants inclusion. For this reason we have decided not to include an `else` keyword.

4.5 Ruleset Definition Language

In order to empower application-level programmers to implement services and applications using the concepts proposed in the previous section, we have designed a ruleset definition language for the FACTS middleware architecture for wireless sensor networks. While several rule-based programming languages do exists [JTC, jes], none of them was designed with a

focus on saving memory when running the programs. This however is a crucial requirement for the FACTS rule engine. Specifically, the programming concepts of slots, which are used to address facts in the fact repository, thereby avoiding local variables, has to the best of our knowledge not been proposed before.

In this section we will introduce the syntax and semantics of the FACTS ruleset definition language and give a few simple examples. For the complete grammar of the language, refer to Appendix A.

4.5.1 Syntax

The FACTS ruleset definition language consists of the following programming constructs:

Rulesets

A ruleset as declared using the `ruleset` keyword is the basic translation unit of the language and implements a self-contained application or service. It is identified by an alphanumerical name and comprises several declaration blocks. Each of these declares either a name, a slot, a rule, a fact, or an external dependency of this ruleset on another ruleset.

Listing 4.1: Example: `ruleset` keyword.

```
1 ruleset myRuleset
2
3 <dependency block>
4 <name block>
5 <slot block>
6 <rule block>
7 <fact block>
```

Each ruleset is saved in its own file whose name typically has the suffix `*.rls`.

External Dependencies

A ruleset may declare dependencies on other rulesets using the `depends` keyword. If ruleset A depends on the services offered by rulesets B and C, then the resulting declaration is as follows:

We plan to extend this mechanism to import the public interfaces of the rulesets for which dependencies have been declared into the current namespace in order to allow direct interaction between rulesets. This will increase the expressiveness of the language to more than the productive coexistence of rulesets that we have implement at present. However,

Listing 4.2: Example: `depends` keyword.

```
1 ruleset myRuleset_A
2
3 depends myRuleset_B
4 depends myRuleset_C
```

we first need to research into exactly which symbols to include into the public interface of a ruleset should be and how to handle namespaces (see Section 8.2).

Names

Complementary to the dependency declarations described above, we plan for names as declared using the `name` keyword to evolve into a building block of the public interface of rulesets. As rulesets process facts, the interaction between different rulesets must be implemented in the form of facts, too. Hence a ruleset must be able to export the names of the facts that it will process for other rulesets to create facts with matching names. The `name` keyword implements this mechanism as shown below:

Listing 4.3: Example: `name` keyword.

```
1 ruleset myRuleset
2
3 name somePublicName = "myInternalName"
```

Facts

In order to allow for declarations of constants or initialization, the `fact` keyword followed by the alphanumerical name of the fact and an optional list of key-variable-tuples to initialize the properties can be used. It adds the specified fact to the fact repository of the sensor node before the rule engine begins execution:

Listing 4.4: Example: `fact` keyword.

```
1 ruleset myRuleset
2
3 fact someFact
4 fact anotherFact [propertyA = 12, propertyB = "test"]
```

In the example above, line 3 adds a simple fact called "`someFact`" without any properties to the repository. Line 4 adds a fact called "`anotherFact`" with two properties, "`propertyA`" is of type `int` and "`propertyB`" is of type `string`.

Note that these facts are *not* tagged as modified and will not fire any rules when the rule engine is run for the first time.

Rules

A rule, being the central building block of a ruleset, is declared using the `rule` keyword followed by the name of the rule and its priority. Higher priority rules are evaluated first during a run of the rule engine. The main body of a rule consists of a list of conditions that describe in which cases the rule should fire, and a list of statements that define the subsequent changes to the fact repository, communication with other nodes, and and interactions with the firmware. Conditions are always prefixed with "<–" and statements are always prefixed with "–>":

Listing 4.5: Example: `rule` keyword.

```
1 ruleset myRuleset
2
3 rule retractSomeFact 100
4 <- exists {someFact}
5 -> retract {someFact}
```

Slots

In the above example there are two blocks encapsulated in curly braces ("{" and "}"). Each of them is a slot whose function is to address facts in the fact repository. A slot consists of the name of the fact, the property key, and a list of conditions. Each of the latter two arguments is optional.

Slots are used very frequently when implementing rulesets. In order to avoid code duplication, it is possible to declare named slots using the `slot` keyword quite similar to the declaration of names described earlier.

Listing 4.6: Example: `slot` keyword.

```
1 ruleset myRuleset
2
3 slot mySlot = {someFact}
4 slot mySlotWithProperty = {someFact propertyA}
5 slot mySlotWithPropertyAndConditions = {someFact propertyA
6   <- <condition>
7   <- <condition>
8 }
```

Line 3 defines a simple slot that matches any fact in the fact repository with the name "`someFact`". The slot defined in line 4 goes beyond that and allows access to one of the property values of the same fact, or to a set of values in case there are multiple facts with matching names. Finally, the slot in lines 5 to 8 addresses the same property, but additionally requires the fact to satisfy two conditions. We will discuss the syntax of conditions in the next section and then return to this example.

Conditions

We have already seen simple conditions in the examples for rules and slots. Always prefixed with "<–" and as introduced in Section 4.2.2, there are two kinds of conditions:

`exists` is followed by a slot and is true if a matching fact is found in the fact repository.

`eval` allows for more complex expressions to be evaluated using comparison and simple arithmetic operators.

Listing 4.7: Example: Simple conditions.

```
1 rule myRule 100
2 <- exists {someFact}
3 <- eval ({someFact propertyA} == true)
4 <- eval (({someFact propertyB} + 12) != {anotherFact propertyB})
5 <- eval ((2 ^ {anotherFact propertyB}) >= (sqrt {yetAnotherFact propertyC}))
6 -> [...]
```

The condition in line 2 simply checks whether a fact named "`someFact`" exists in the fact repository. The condition in line 3 is true if "`propertyA`" of "`someFact`" is of type `bool` and has the value `true`. Line 4 checks whether "`propertyB`" of "`someFact`" incremented by 12 is not equal to "`propertyB`" of "`anotherFact`", and line 5 checks whether 2 to the power of "`propertyB`" of "`anotherFact`" is greater or equal than the square root of "`propertyC`" of "`yetAnotherFact`".

Furthermore, there are several additional operators that specifically make use of slots being resolved to a set of facts:

`count` counts how many facts matching the following slot exist in the fact repository.

`sum` adds the values of the respective property of all facts that match the slot. If no fact matches the slot the result is 0.

`product` multiplies the values of the respective property of all facts that match the slot. If no fact matches the slot the result is 1.

min returns the minimum value of the set of the respective property of all facts that match the slot. If no fact matches the slot the result is undefined.

max returns the maximum value of the set of the respective property of all facts that match the slot. If no fact matches the slot the result is undefined.

Note that the latter four operators only apply to properties of type `int`.

Listing 4.8: Example: Complex conditions.

```
1 rule myRule 100
2 <- eval ((count {someFact}) == 1)
3 <- eval ((sum {someFact propertyA}) >= 12)
4 <- eval ((product {someFact propertyA}) > 12)
5 <- eval ((min {someFact propertyA}) < 12)
6 <- eval ((max {someFact propertyA}) <= 12)
7 -> [...]
```

Slots Revisited: `this`

Now that we have introduced both slots and conditions (whose definitions are mutually recursive), we will explain how slots and conditions interact. Coming back to the slot example presented earlier, we can now declare proper conditions for the slot:

Listing 4.9: Example: `this` keyword.

```
1 slot mySlotWithPropertyAndConditions = {someFact propertyA
2   <- eval ({this propertyA} == "test")
3   <- eval ({this propertyB} > {anotherFact propertyC})
4 }
```

This example introduced the `this` keyword. It is used to refer to the one single fact that the rule engine is currently matching against the slot. As part of this matching procedure, the rule engine will access other facts in the fact repository. Hence, the fact currently being matched is first removed from this "working set" as otherwise the condition in line 2 would always evaluate as true as the fact is simply matched against itself.

Note that trying to implement the same semantics without `this` will *not* work as expected:

As there may be multiple facts with the name "`someFact`" in the fact repository, line 2 may evaluate as true for one fact and line 3 may evaluate as true for a different fact. This would result in the slot matching facts that the programmer most probably did not intend to match.

Listing 4.10: Example: Slot without `this` keyword.

```
slot mySlotWithPropertyAndConditions = {someFact propertyA
  <- eval ({someFact propertyA} == "test")
  <- eval ({someFact propertyB} > {anotherFact propertyC})
}
```

It is important to always keep in mind that a slot may match multiple facts in the fact repository and that the resulting set of facts will be operated on as described in Section 4.4.1. It is up to the programmer to carefully design slots to match exactly those facts that he intends to.

Statements

The missing brick for a complete ruleset definition are statements. Available statements have already been introduced in Section 4.2.2. In the ruleset definition language they are always prefixed with "->" and their syntax is a follows:

Listing 4.11: Example: Statements

```
rule testAllStatements 100
<- exists {someTriggeringFact}
-> define "someFact"
-> define "anotherFact" [propertyA = true, propertyB = 12]
-> set {someFact propertyA} = false
-> set {someFact propertyB} = ({anotherFact propertyB} * 12)
-> send <toMAC> <txPower> {someFact}
-> copy {someFact}
-> flush {someFact}
-> retract {someFact}
-> call someFunction ({anotherFact propertyA}, {anotherFact propertyB})
```

In the example above, we have introduced the syntax of the following statements:

define creates a new fact in the fact repository, optionally with properties. The fact is tagged as modified in the next run of the rule engine.

set updates an old or adds a new property to a fact and assigns a value to it.

send copies one or multiple facts over the network. "*<toMAC>*" and "*<txPower>*" are expressions that define the addressee and the transmit power to use. Both values depend either on the hardware platform or the deployment scenario.

copy makes an exact copy of the facts specified by the following slot. The ownership and modification time of the new fact are updated, and it is tagged as modified.

flush removes the modified flag from the matching facts so that they will not cause rules to fire during the next run of the rule engine.

retract removes the facts matching the given slot from the fact repository.

call invokes a function with parameters as defined by the list of expressions in order to access the firmware directly or perform some resource critical computation.

For readability we have skipped a few minor details of the ruleset definition language. They are however quite self-explanatory and will be introduced as part of the examples in the next section. Again, refer to Appendix A for the complete grammar of the language.

4.5.2 Examples

Now that syntax and semantics of the FACTS ruleset definition language have been introduced, we provide a few simple examples to illustrate how the different concepts work together. Refer to Chapter 7 for an in-depth discussion of a more complex example.

A Simple Timer Ruleset

The Simple Timer ruleset shows how two rules interact with each other by generating a fact for the other rule to react to. As we are dealing with time in this example, the facts are called `tick` and `tack`. Along with them, there is a `counter` fact that counts the occurrences of `tick` facts. The complete ruleset is shown in Listing 4.12.

Listing 4.12: A simple timer ruleset.

```
1  ruleset Timer
2
3  slot counterTicks = {"counter" ticks}
4
5  fact "counter" [ticks = 0]
6
7  rule tick 100
8  <- exists {"tack"}
9  -> retract {"tack"}
10 -> define "tick"
11 -> set counterTicks = (counterTicks + 1)
12
13 rule tack 100
14 <- exists {"tick"}
15 -> retract {"tick"}
16 -> define "tack"
```

After the ruleset declaration in line 1, a slot referring to the `ticks` property of the `counter` fact is defined in line 3. It is used for easy access to that property. In line 5, the `counter` fact is created and initialized, with its `ticks` property set to 0. The core of the ruleset are the two following rules, which are almost identical. Each of them fires if a `tack` or respectively a `tick` fact has been created in the previous run of the rule engine. The rules then proceed to remove this fact and create a new `tick` or `tack` fact for the respective other rule to fire in the next run. Additionally, the `tick` rule increments the `ticks` property of the `counter` fact, as addressed by the `counterTicks` slot in line 11.

There are two noteworthy issues about this example: In order to activate this ruleset, either a `tick` or a `tack` fact must be provided externally, possibly as a reaction to an event that occurred on the sensor node. More important however is the issue that this ruleset creates a cyclic interaction between two rules. While intentional in this example, cyclic interactions should generally be avoided as they may result in infinite loop that undermine the event-driven semantics, have the potential to destabilize a distributed rule-based algorithm, and generally waste energy. We plan to add tool support for the detection of cyclic interaction between rules in the future (see Section 8.2).

Temperature Monitoring Ruleset

The Temperature Monitoring example while still quite simple draws more towards realistic WSN applications: The ruleset implements the distribution of sensor readings, in this case temperature values, above a certain threshold value. Listing 4.13 presents the complete rulesets.

The ruleset begins with defining a public name for the `temperature` fact in line 3, in order to allow other rulesets to process the data once it has been distributed. Lines 5 and 7 are a common idiom in the ruleset definition language to retrieve the ID of the sensor node that the current instance of the rule engine is running on. We consider to provide a standard set of facts that hold this kind of node-specific information as part of a system-level ruleset in the future.

The main processing of the ruleset is done in the `readAndSend` rule, which reacts to `temperature` facts that have been generated on the local sensor node, i.e. the owner of the `temperature` fact is equal to the node ID, and whose `value` property matches certain conditions. In this case, the `value` property of the `temperature` fact simply needs to be greater or equal than a threshold value of 20 for the rule to fire. Once fired, the rule sends the `temperature` fact that fired the rule to all neighboring nodes (0 is the broadcast MAC address) at a transmit power of 15.[1] Retrieving the `temperature` fact

[1] These are arbitrary numeric values in the current example. For better integration with the hardware platform we plan to add system-wide constants in the future.

Listing 4.13: Temperature monitoring ruleset.

```
1  ruleset Temperature
2
3  name temperature = "temperature"
4
5  fact "node"
6
7  slot nodeID = {"node" owner}
8  slot myNewTemperature = {temperature
9    <- eval ({this owner} == nodeID)
10   <- eval ({this modified} == true)
11 }
12
13 rule readAndSend 20
14 <- exists {temperature
15   <- eval ({this owner} == nodeID)
16   <- eval ({this value} >= 20)
17 }
18 -> send 0 15 myNewTemperature
```

that is to be transmitted is handled by the `myNewTemperature` slot. The fact needs to satisfy two conditions: It must be owned by the local node to avoid retransmitting facts that have been received from other nodes, and it must be tagged as modified to identify it as the fact that has fired the rule as opposed to older temperature readings.

Once again, there are two noteworthy issues about this ruleset: For one, it shows how seamlessly the system-generated `owner` and `modified` properties integrate with the application level code. More importantly, the definition of the `myNewTemperature` slot in lines 8 to 11 illustrates the importance of properly designing the slots: If the `modified` condition had been omitted, then each firing of the `readAndSend` rule would broadcast *all* `temperature` facts from the local fact repository over the network, which is far from the intention of the ruleset.

Coverage Ruleset

The goal of a coverage algorithm in a wireless sensor network scenario is to determine which areas of a geographic region are covered by the sensor network. The information gained may be used to selectively power down sensor nodes that are redundant in order to extend the total lifetime of the sensor network. A partial listing of the rule-based implementation of the coverage algorithm is given in Listing 4.14. The complete ruleset can be found in Section B.1.

As a first step and firing in the initialization state, the `range` rule removes the `init` fact that caused it to fire from the fact repository in line 7, and sets the node's state to

Listing 4.14: Coverage ruleset (excerpt).

```
ruleset Coverage

[...]

rule range 50
<- exists {init}
-> retract {init}
-> set {"node" state} = "ON"
-> define "range" [xMin = (nodePosX - 10), xMax = (nodePosX + 10), yMin = (
    nodePosY - 10), yMax = (nodePosY + 10)]
-> send {system broadcastMAC} {system txPower} {"range"}

[...]

slot rangeXMinYMinOwner = {"range" owner
  <- eval ({this owner} != nodeID)
  <- eval ({this xMax} >= nodePosX)
  <- eval ({this yMax} >= nodePosY)
  <- eval ({this xMax} <= (nodePosX + 10))
  <- eval ({this yMax} <= (nodePosY + 10))
}

[...]

rule xMinYMinCovered 30
<- eval ({node state} == "ON")
<- exists rangeXMinYMinOwner
-> retract {"coveredXMinYMin"
  <- eval ({this byNode} == rangeXMinYMinOwner)
}
-> define "coveredXMinYMin" [byNode = rangeXMinYMinOwner]

[...]

rule determineCoverage 40
<- exists {"coveredXMaxYMax"}
<- exists {"coveredXMaxYMin"}
<- exists {"coveredXMinYMin"}
<- exists {"coveredXMinYMax"}
-> define "covered"
-> send {system broadcastMAC} {system txPower} {"covered"}
-> set {node state} = "OFF"
```

`"ON"` in line 8. It then proceeds to calculate the range it expects to cover and stores this information in a `range` fact in line 9. Note that establishing the position of the node is not in the scope of the coverage algorithm. In line 10 the newly created `range` fact is broadcasted to the neighboring nodes.

In the next stage, the node waits for the `range` facts of its neighbors. Upon reception of a matching fact, the `xMinYMinCovered` rule inspects the data in line 26 using the `rangeXMinYMinOwner` slot defined in lines 14 to 20 and fires if the covered area as reported by the `range` fact overlaps with its own. The result is stored in the `coveredXMinYMin` fact that also holds the information which node overlaps in the given direction. Similar rules for the `coveredXMaxYMin` fact, `coveredXMaxYMax` fact, and `coveredXMinYMax` fact have been omitted for brevity.

Finally as a last step, the `determineCoverage` rule checks whether all four sides of a nodes area are covered by other nodes in lines 35 to 38, and if this is the case, stores this information in the `covered` fact, broadcasts this once again and turns the sensor node off afterwards. After this process has been completed for all nodes, each node knows whether it is the only node of the wireless sensor network to cover one particular geographic region or not and can act accordingly in the future.

The example of the coverage algorithm illustrates how our middleware provides intuitive event-like semantics and abstraction from low-level communication details. Furthermore, it shows how remote data transparently becomes available for local processing while still preserving the semantics of a globally shared information space.

Now that concepts and language have been covered in detail and illustrated with a few examples, we will move on to describe implementation details, such as the compilation process, bytecode layout, and execution of rulesets, in the next section.

Chapter 5

Compilation and Execution

After describing in detail the conceptual aspects and introducing the ruleset definition language of the FACTS middleware architecture for wireless sensor networks in the previous chapter, we now turn to the more practical issues. The following subprojects have been implemented successfully, or, as for the last subproject, the implementation is ongoing:

FACTS-rc is the ruleset compiler for the FACTS middleware architecture, that compiles ruleset definitions into one of two possible output formats. Its frontend is implemented using `lex` and `yacc` definitions derived in part from the type declarations in FACTS-hs (see below). It has two backends implemented in C, one to generate Haskell definitions as input for FACTS-hs, and one to generate optimized bytecode for interpretation by FACTS-re (once again, see below).

FACTS-hs is an implementation of the FACTS middleware architecture and additionally offers a rudimentary simulation environment of a wireless sensor network. Implemented in the functional and strongly typed Haskell programming language, its goal is to serve as a prototype while establishing the concepts as discussed in Chapter 4, to provide a reference for future implementations, and to function as a testing environment for rulesets.

FACTS-re is an implementation of the FACTS runtime environment on the ScatterWeb platform. It is implemented in C and runs as a user application on the ScatterWeb sensor nodes. At the time of this writing, it does not yet implement the entire functionality offered by FACTS.

In this chapter we will discuss each of these components separately: Section 5.1 covers FACTS-rc and how applications written in the form of rulesets are compiled into bytecode. Section 5.2 and Section 5.3 describe how the bytecode is executed on our functional prototype implementation and on the ScatterWeb implementation of the FACTS runtime environment respectively.

5.1 The Compilation Process: FACTS-rc

In order to run applications written in the FACTS ruleset definition language, they first need to be compiled. To this end, we have implemented the FACTS-rc ruleset compiler. As already mentioned above, its frontend is implemented in `lex` and `yacc` definition files for lexing and parsing. The parsing code in particular was derived from the functional type declarations of the FACTS-hs subproject.

The parser creates an internal data structure that can be used as input for the two backends of the compiler. One backend produces Haskell definitions that serve as input to the FACTS-hs middleware simulation. It generates one Haskell file per ruleset and FACTS-hs takes care of dependency resolution. The other backend produces bytecode for interpretation by the FACTS-re runtime environment on a ScatterWeb sensor node. Several ruleset may be compiled into one bytecode image, which then is added to a ScatterWeb EEPROM image for deployment on the sensor nodes. The FACTS-re backend of the compiler also takes care of reducing the size of the generated bytecode as much as possible through optimization.

In the following, we will illustrate the implementation details of both the parsing and the optimization processes by the means of an example.

5.1.1 Example: Turing Machine Ruleset

Listing 5.1 gives an excerpt of the implementation of a Turing Machine in the FACTS rule definition language. We have chosen this example as the focus of this section is primarily on parsing and bytecode optimization rather than applicability to the wireless sensor network domain. On the other hand, having a complete Turing Machine implementation, consisting of merely four rules and running on our middleware architecture nicely illustrates the expressiveness of the language and the power of the underlying concepts.

In order to provide a better understanding, let us go over the short excerpt of the Turing Machine ruleset (see Listing 5.1): A unique `run` fact as accessed by the slots in lines 9 and 10 holds the current state of the Turing Machine and the current position of the read/write head on the tape. Both the tape and the transition function are implemented as sets of facts, one of which is shown in Figures 5.1 and 5.2 respectively.

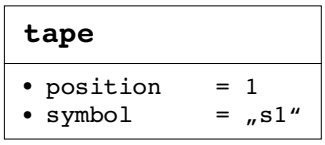

Figure 5.1: A `tape` fact as used by the Turing Machine ruleset.

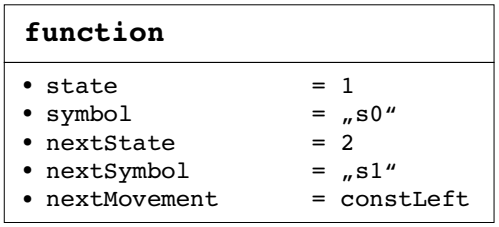

Figure 5.2: A `function` fact as used by the Turing Machine ruleset.

Listing 5.1: Turing Machine ruleset (excerpt).

```
1  ruleset TuringMachine
2
3  name run = "run"
4  name function = "function"
5  name tape = "tape"
6
7  [...]
8
9  slot runState = {run state}
10 slot runPosition = {run position}
11 [...]
12
13 slot currentTapeSymbol = {tape symbol
14   <- eval ({this position} == runPosition)
15 }
16
17 slot functionNextState = {function nextState
18   <- eval ({this state} == runState)
19   <- eval ({this symbol} == currentTapeSymbol)
20 }
21 slot functionNextSymbol = {function nextSymbol
22   <- eval ({this state} == runState)
23   <- eval ({this symbol} == currentTapeSymbol)
24 }
25 slot functionNextMovement = {function nextMovement
26   <- eval ({this state} == runState)
27   <- eval ({this symbol} == currentTapeSymbol)
28 }
29
30 [...]
31
32 rule step 100
33 <- exists {run}
34 -> define "next" [state = functionNextState, symbol = functionNextSymbol,
       movement = functionNextMovement]
35 -> set runState = {"next" state}
36 -> set currentTapeSymbol = {"next" symbol}
37 -> set runPosition = (runPosition + {"next" movement})
38 -> retract {"next"}
```

A `tape` fact describes one cell of the tape at a certain `position` and containing a `symbol`. A `function` fact describes which `nextState`, `nextSymbol`, and `nextMovement` follow from a given `state` and a given `symbol`. Note that there must be a `function` fact for all possible combinations of states and symbols used by the Turing Machine, as otherwise the definition of the transition function would be incomplete. Slots to conveniently access one particular fact of the set of facts that make up either the tape or the transition function are defined in lines 13 to 28.

The core of the Turing Machine ruleset is the `step` rule defined in lines 32 to 38. It fires as long as the `run` fact exists and has been modified in the previous run of the rule engine. The first statement of the rule in line 34 defines a new temporary fact that stores the next state, symbol and movement based on the data retrieved from the set of `function` facts using the respective slots introduced above. The properties of the temporary fact are then used to update the `state` and `position` properties stored in the `run` fact and the tape symbol at the position of the read/write head as addressed by the `currentTapeSymbol` slot (lines 35 to 37). Afterwards, the temporary fact is retracted.

Only the parts relevant to this example have been discussed in detail. Additional rules contained in the complete listing in Section B.2 implement error handling and automatically extending the tape by defining new `tape` facts when the read/write head reaches the border of the existing tape.

5.1.2 Parsing and Bytecode Data Structure

The frontend of the FACTS-rc compiler parses the Turing Machine ruleset from Listing 5.1 into an abstract syntax tree (AST) as shown in Figure 5.3. Based on this, the generation of Haskell code for FACTS-hs is a trivial syntactical transformation that we do not discuss in detail. Instead we concentrate on how the bytecode for the FACTS-re runtime environment is generated.

The data structure of the bytecode is shown in Figure 5.4. For readability all constants have been spelled out and identifiers, i.e. fact names and property keys, have been preserved. In the actual data structure they are converted into unique numerical identifiers. For this example, Figure 5.1 shows the strings and their corresponding encoded bytecode identifiers. There are several differences between the data structure and the abstract syntax tree it is derived from:

- As one bytecode image may contain several rulesets, the ruleset information is omitted. The FACTS-re runtime environment deals directly with rules and facts. When necessary, the separation between rulesets is preserved by retaining the namespace scope of names of facts.

- All named items of the ruleset, i.e. names and slots, have been copied into the locations of the data structure at which they have been referenced. While this actually increases the size of the data structure at the moment, it later avoids special cases in the optimization algorithm.

- All list structures are removed as they incur additional overhead for storing pointers. Instead we organize items in several large arrays, one for each type of item. This organization of data has some peculiarities which we discuss below.

In order reduce the size of the bytecode image, we try to avoid pointers in data structures as much as possible: Instead of going with the obvious choice of using linked lists for storing the complex data structures that make up a ruleset, we sort all our data structures by their type and then access them via pointer arithmetic, i.e. in an array-like fashion.

This technique avoids the storage overhead of linked lists, which is linear in the elements of the list due to the pointers contained in each element of the list structure, and replaces it with a constant overhead of two pointers, one pointing to the first and one pointing to the last element of the list. Alternatively, we could have used one pointer to the first element and one integer to specify how many of the following elements belong to the list. However, this has a small drawback when it comes to implementing the rule engine: A fragment commonly found in the code of the rule engine is that of iterating over a set of elements in the bytecode. If we had represented a list with a pointer and an integer, a loop over the elements of a list, which for this example we assume to be of type `fact_t`, would have looked as shown below.

Listing 5.2: List iteration with start address and length parameters.

```
1 UINT16 fact_addr, i;
2 for(i = 0; i <= header.facts_count; i++) {
3   fact_addr = header.facts_start + (i * sizeof(fact_t));
4   // Now retrieve fact at fact_addr and process it.
5 }
```

With our current representation of lists, the same loop can be implemented as follows:

Listing 5.3: List iteration with start and end address.

```
1 UINT16 fact_addr;
2 for(fact_addr = header.facts_first; fact_addr <= header.facts_last;
3           fact_addr += sizeof(fact_t)) {
4   // Now retrieve fact at fact_addr and process it.
5 }
```

Note that the second implementation only uses one automatic variable while the first implementation uses two of them. The resulting reduction of memory usage may seem negligible, however when taking into account the two points that iterations over lists are very common in the implementation of the rule engine, and that these loops are deeply nested in most cases, the representation with two pointers is clearly superior.

Note that this scheme only works because, data structures are sorted by type even if this places data items that are logically related far away from each other in real memory. The theoretical drawback is that data looses locality, which may result in poor performance if techniques such as caching memory are used. However, this can be mitigated by adapting the caching algorithm.

Taking a step back, it is obvious that there is still a lot of redundancy present in the data structure shown in Figure 5.4. In the following section we will take care of removing all this redundant information.

Identifier	**Bytecode Value**
`"run"`	0x02
`"state"`	0x03
`"constants"`	0x04
`"errorState"`	0x05
`"position"`	0x06
`"leftBorder"`	0x07
`"tape"`	0x08
`"symbol"`	0x09
`"blankSymbol"`	0x0A
`"rightBorder"`	0x0B
`"next"`	0x0C
`"function"`	0x0D
`"nextState"`	0x0E
`"nextSymbol"`	0x0F
`"movement"`	0x10
`"nextMovement"`	0x11
`"right"`	0x12
`"left"`	0x13
`"neutral"`	0x14

Table 5.1: Encoded bytecode identifiers of the Turing Machine ruleset.

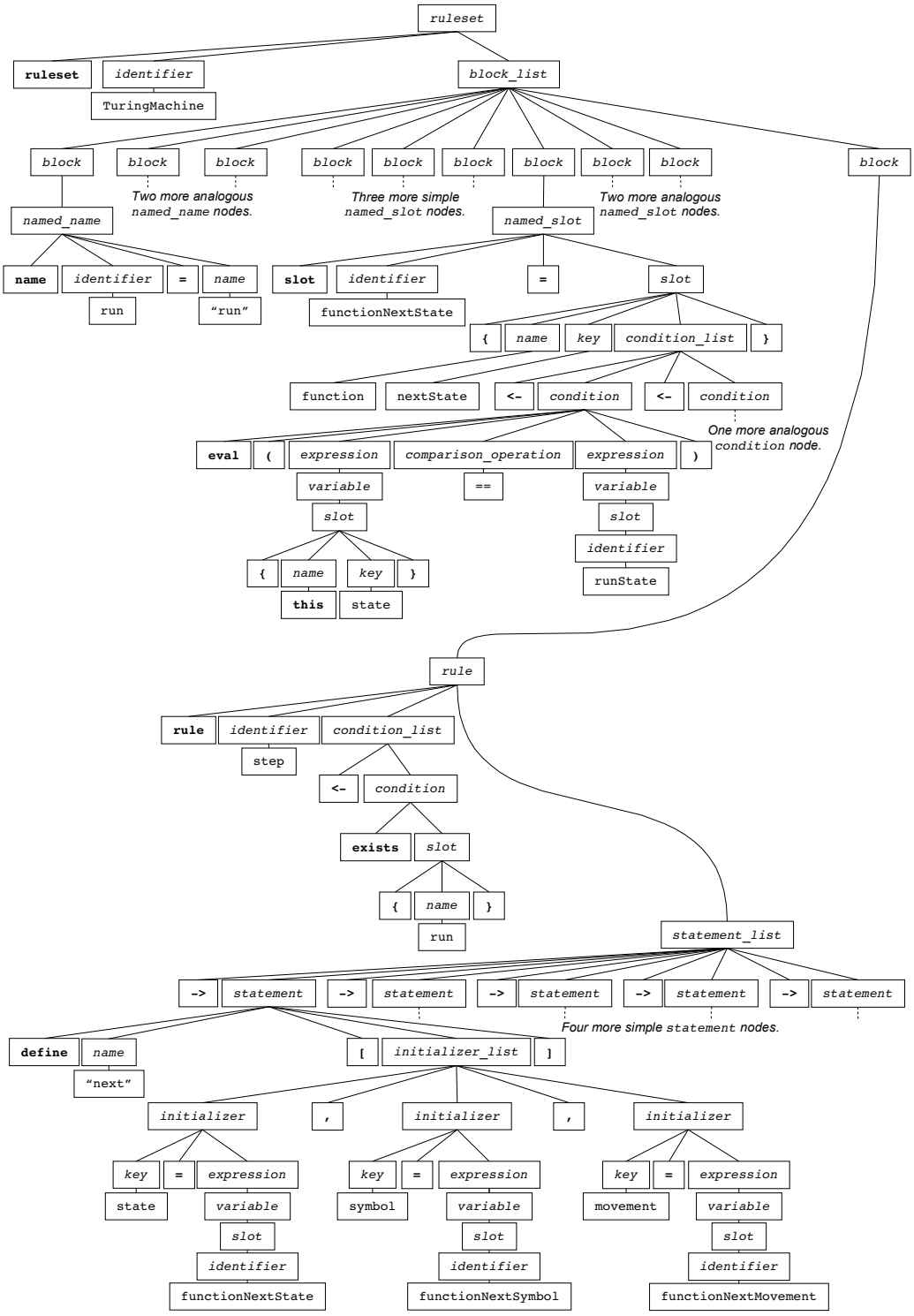

Figure 5.3: Abstract syntax tree of Listing 5.1. Some redundant parts have been omitted for readability. Non-terminals are printed in italics, language keywords are printed in bold.

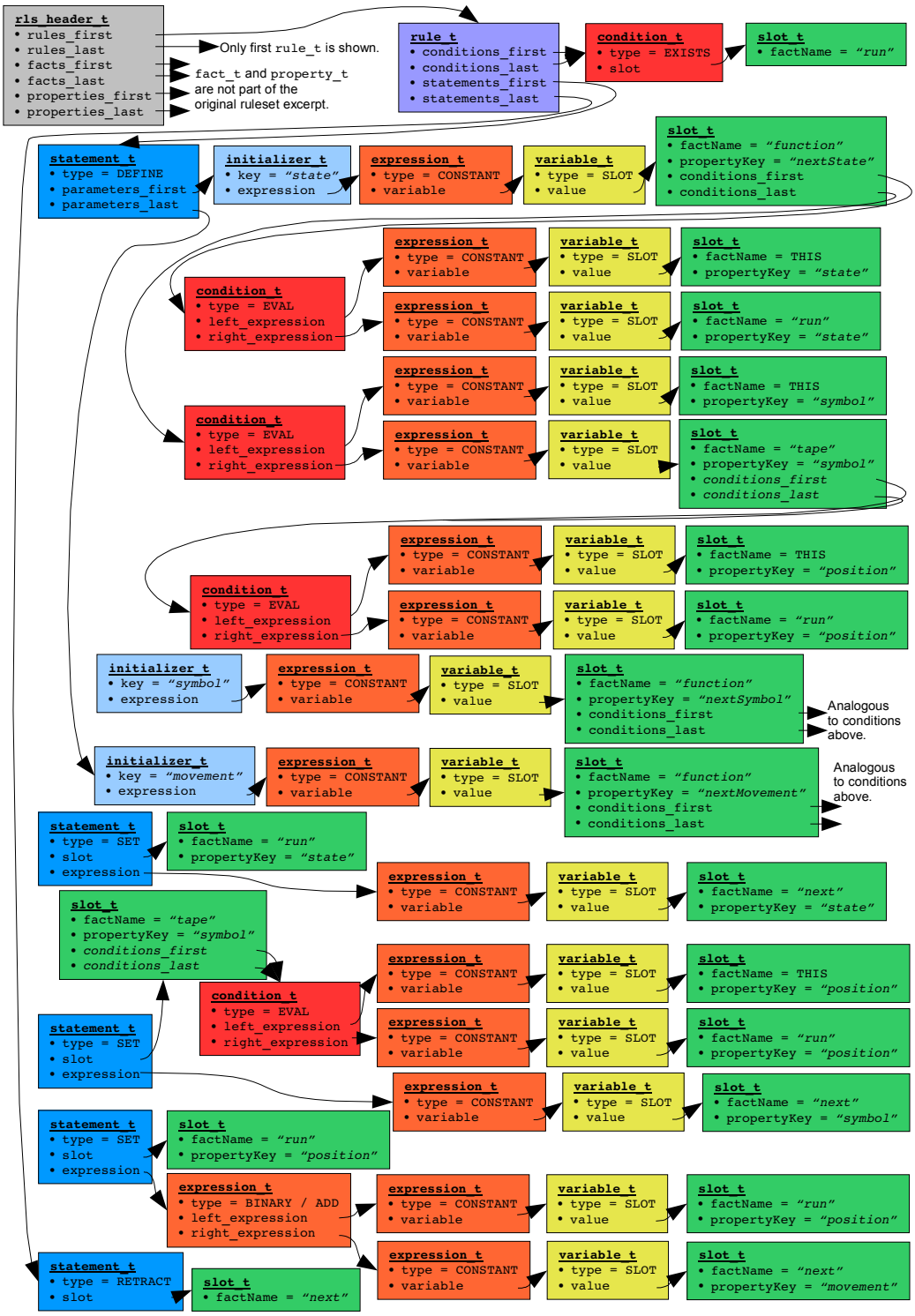

Figure 5.4: Bytecode data structure derived from the AST in Figure 5.3. Some redundant parts have been omitted for readability. Note how similar data is duplicated in several places within the data structure.

5.1.3 EEPROM Memory Layout and Bytecode Optimization

As already mentioned in the previous section, data structures of identical type are organized as arrays in the bytecode to save the structural overhead of linked lists. The regions for the different types are laid out in the EEPROM bytecode image as shown in Figure 5.5.

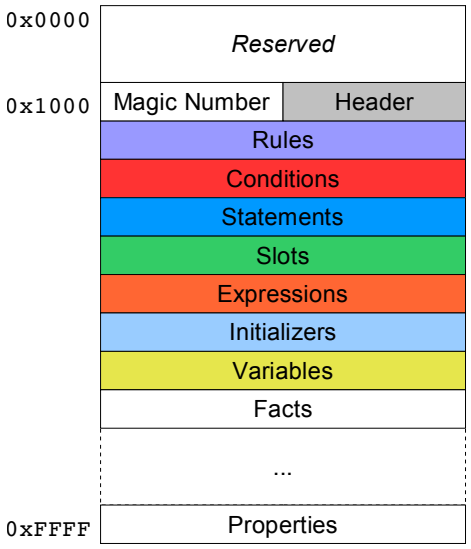

Figure 5.5: Bytecode layout within the EEPROM of a ScatterWeb sensor node.

On a ScatterWeb sensor node the EEPROM from addresses `0x0000` to `0x0FFF` is reserved for the firmware. Data of user applications begins at address `0x1000`, in our case with a 16 bit magic number to identify the following data to be a FACTS bytecode image. After the unique bytecode header, the EEPROM memory is partitioned into regions, one for each bytecode data type. Within these regions structures of identical type are densely packed. Except for the gap between the fact and the property regions, no memory is wasted between the regions.

The fact and the property regions are special as they are the only regions that will change in size at runtime as facts and their properties are added to the fact repository. The fact region grows up in memory, while the property region starts at address `0xFFFF` and grows down in memory. It is up to the FACTS-re rule engine to make sure that the regions never overlap.

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 to be performed in order to reduce the size of the bytecode image.

The FACTS-rc ruleset compiler parses rulesets implemented in the ruleset definition language and generates a tree-like data structure with its root in the `rls_header_t` structure to be used in the bytecode. For the Turing Machine example we have already shown this data structure in Figure 5.4 and pointed out that it has several redundancies.

The key to optimize the data structure is to find identical subtrees, merge them into one, and adjust pointers in the remaining data structure accordingly. Once this process is complete, the resulting data structure has lost its tree property but still is a directed acyclic graph (DAG). Figure 5.6 shows the optimized data structure for the Turing Machine example.

There are two difficulties to take into account when implementing the optimization algorithm:

- Optimization is recursive: Consider two subtrees A and B with their root elements a and b. Initially, a and b are not identical because the respective subtrees are located at different EEPROM memory addresses and hence the pointers in a and b do not match. After completely generating the elements pointed to by b, it is however established that they are identical to those generated previously when working on subtree A. This results in the fields in b being updated to point to the elements contained in A, which in turn renders a and b identical. Merging a and b however requires all external references to b to be updated to point to a instead.

- The results of the optimization need to be known before the bytecode generation can start: As the elements of the bytecode contain EEPROM addresses, the location of all elements must be known when generating them. At the same time the bytecode is densely packed in the EEPROM memory, not wasting any space between the regions for each data type. Together, these two constraints result in the requirement of knowing the exact number of elements per type before generating the bytecode. However, as optimization is part of the generation process, we have a chicken-egg-problem.

In order to solve the optimization problem, bytecode generation is implemented as a two phase process: In the first phase, the bytecode backend of the FACTS-rc ruleset compiler generates bytecode assuming the worst case, in which no elements of the bytecode are redundant, and allocates the regions for the data types accordingly. The resulting temporary bytecode image is not optimal in terms of size because it has gaps between the used regions of each data type. However, in this first phase the compiler keeps track of how many elements of each type are actually generated. In the second phase this information is used to generate an optimal layout of regions of data types into which the elements can be placed without wasting memory.

A typical function that creates a bytecode data structure, in this case of type `condition_t`, is shown in Listing 5.4. It takes two arguments: `rc_condition` is a pointer to a representation of the condition internal to the compiler, and `in_list_context` states whether the current condition is to be generated as part of a list of conditions as opposed to just a single conditions. The difference is that single conditions may be optimized, while conditions generated as part of a list must preserve the ordering within the list as mandated by our addressing scheme for lists.

Line 2 calculates the EEPROM address at which to generate the new `condition_t` based on the start of the region for conditions and the number of conditions generated so far. Line 3 allocates memory for generation of a temporary condition. Line 5 actually writes the content of `rc_condition` into the temporary EEPROM condition structure; the details of this process are omitted for brevity.

Listing 5.4: FACTS-rc function to generate a `condition_t` structure in the EEPROM bytecode image.

```
1  UINT16 eeprom_gen_condition(rc_condition_t* rc_condition, int in_list_context)
       {
2    UINT16 eeprom_address = conditions_first + (conditions_used++ * sizeof(
         condition_t));
3    condition_t* condition = (condition_t*) malloc_zero(sizeof(condition_t));
4
5    eeprom_gen_condition_at(rc_condition, condition);
6
7    if(optimize_bytecode && !in_list_context) {
8      // Optimize bytecode by reusing previously written items that
9      // are exact duplicates of the current item.
10     void* addr = memfind(condition, eeprom_image + conditions_first,
           eeprom_image + conditions_first + ((conditions_used - 1) * sizeof(
           condition_t)), sizeof(condition_t), 1);
11     if(addr) {
12       free(condition);
13       conditions_used--;
14       return (UINT16) (addr - ((void*) eeprom_image));
15     }
16   }
17
18   memcpy(eeprom_image + eeprom_address, condition, sizeof(condition_t));
19   free(condition);
20   return eeprom_address;
21 }
```

Optimization is performed beginning in line 10 by searching the region of previously generated conditions for a `condition_t` that exactly matches the condition we have just generated. If this is the case, line 12 to 14 free the temporary condition, adjust the counter of used conditions, and return the EEPROM address of the previously generated condition. If the current condition has not been generated before, it is added to the bytecode image in line 18 and the new EEPROM address is returned.

The resulting bytecode is shown in Figure 5.7. For readability only the elements that are part of the excerpt of the the Turing Machine ruleset are highlighted, the remaining elements have been left blank. While the data structure is identical to one given in Figure 5.6, it is clearly intended for machine processing rather to be readable by humans. From the coloring it is however obvious that the regions for data types found in the real bytecode correspond to those given in the more abstract memory layout diagram in Figure 5.5.

This concludes the qualitative discussion of FACTS bytecode optimization. We will give a quantitative analysis in the next section.

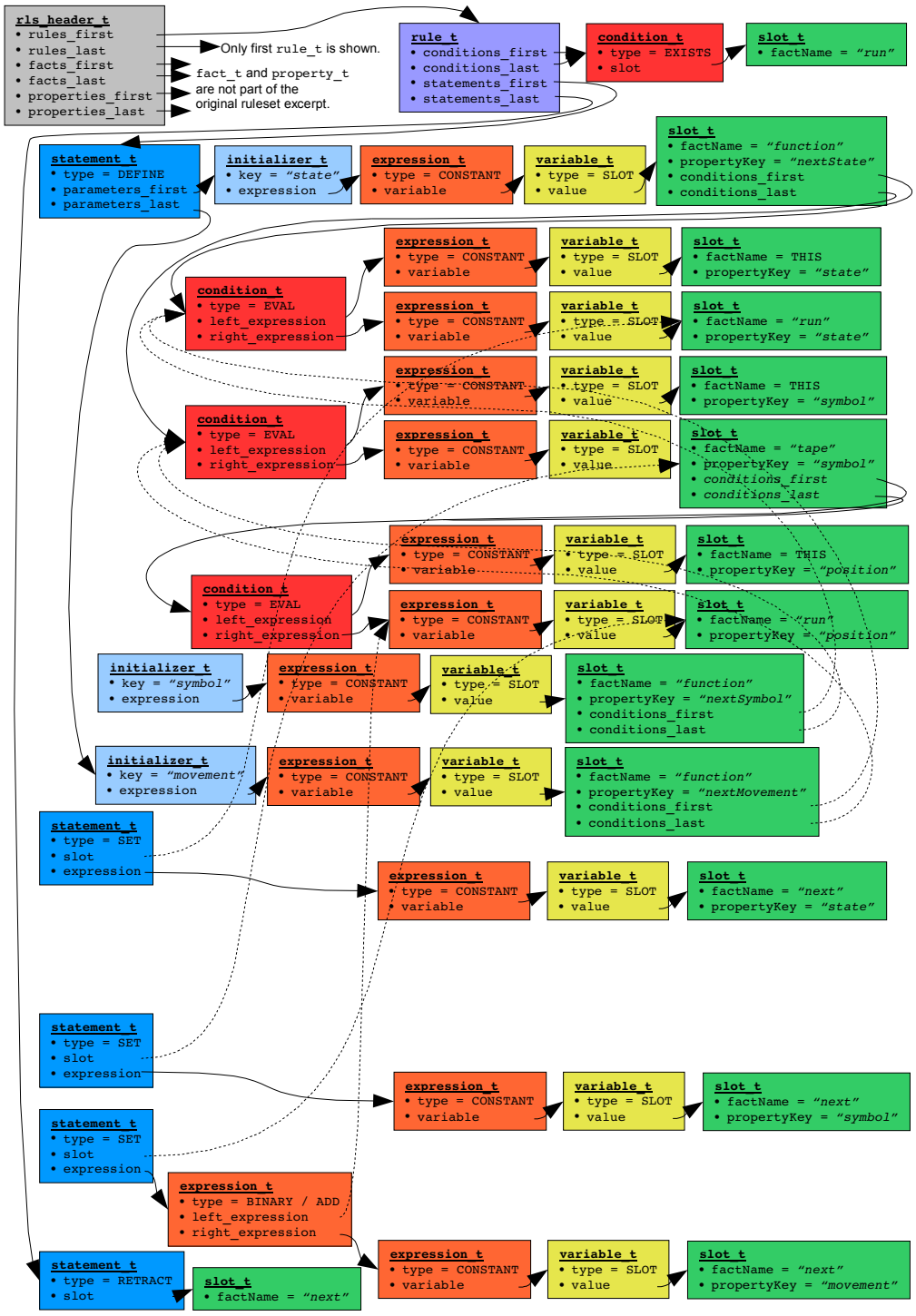

Figure 5.6: Optimized bytecode data structure derived from the data structure in Figure 5.4. Redundant parts have been omitted for readability. Duplicate data has been removed by reusing previously generated items in the data structure as indicated by the dashed arrows.

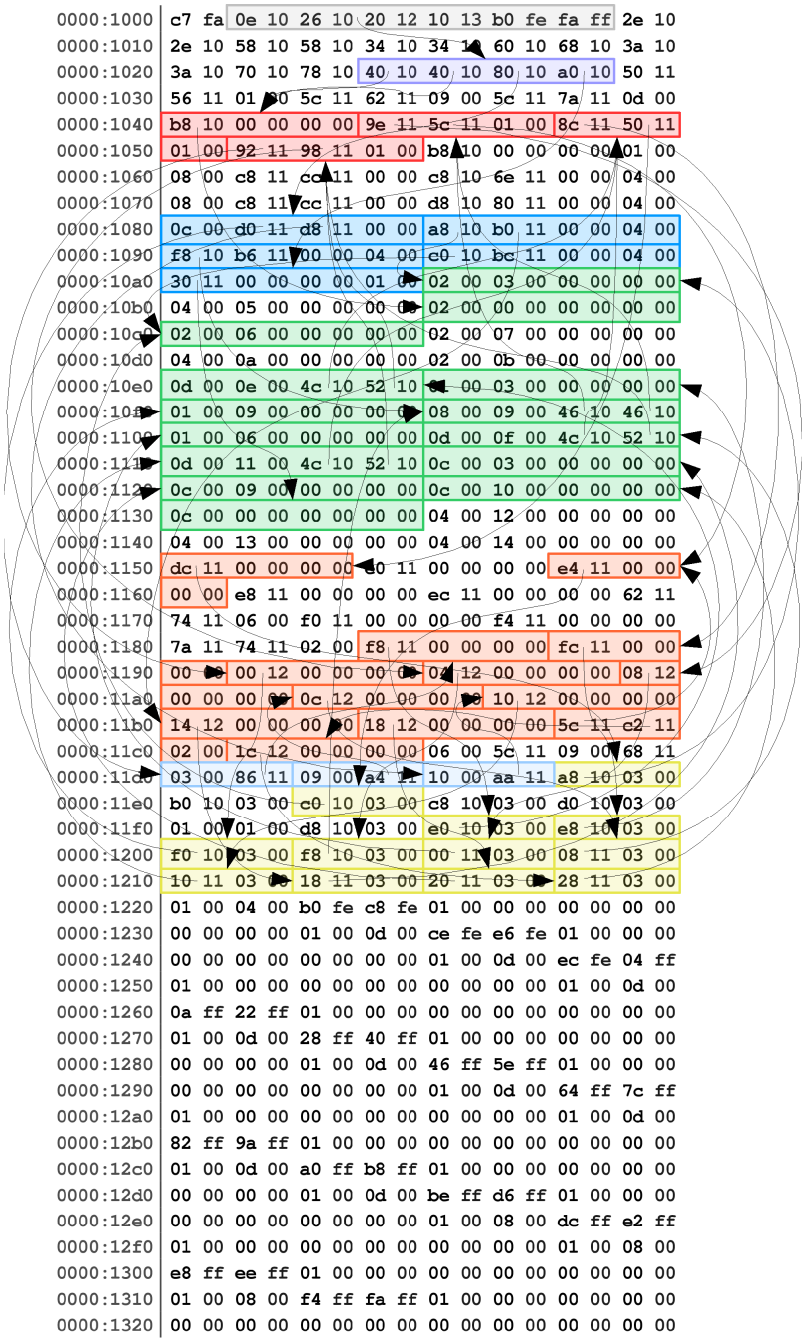

Figure 5.7: Bytecode for the Turing Machine ruleset. The highlights and pointer structure correspond to the structure given in Figure 5.6. The identifiers have been encoded according to Table 5.1. The areas of the bytecode that are not highlighted have no correspondence in the excerpt of the Turing Machine ruleset as given in Listing 5.1, specifically the data beyond address `0x1220` is the encoded fact repository. Finally, the property elements – which are located between addresses `0xFEB0` and `0xFFFF` – have been omitted for brevity.

5.1.4 Evaluation of Bytecode Optimization

After discussing the process of bytecode optimization in detail in the previous section, it is now time to examine the impact of this process. In order to do so, we have collected numbers from three major applications: the Coverage ruleset as covered in Section 4.5.2, the Turing Machine ruleset as covered in Section 5.1.1, and the Generic Role Assignment ruleset that will be the topic of Chapter 7.

For each of these rulesets, Tables 5.2, 5.3 and 5.4 show the item counts and bytes used of the unoptimized and optimized rulesets. Rulesets are broken down into the types of data structures which they contain. For each type of data structure as well as for the ruleset as a whole the total amount of memory used is given before and after the optimization. Further, the percentage saved by optimization is calculated.

The first single item to catch the eye in these tables are the immense savings achieved for the Coverage and the Generic Role Assignment rulesets. 50.4 % and 60,2 % respectively may seem unrealistic at first glance. In order to understand how these savings are possible, it makes sense to look at the reduction in elements used for each type of data structure seperately. Figure 5.8 illustrates the savings for all three rulesets split up by data type.

	Coverage Bytecode				
	unoptimized		optimized		
	#	size	#	size	% saved
Rules	7	56 B	7	56 B	0 %
Conditions	43	258 B	40	240 B	6.9 %
Statements	22	176 B	22	176 B	0 %
Slots	66	528 B	33	264 B	50.0 %
Expressions	107	642 B	24	144 B	77.6 %
Initializers	8	32 B	8	32 B	0 %
Variables	95	280 B	20	80 B	78.9 %
Facts	2	40 B	2	40 B	0 %
Properties	3	18 B	3	18 B	0 %
Total		**2,144 B**		**1,064 B**	**50,4 %**

Table 5.2: Savings achieved by bytecode optimization of the Coverage ruleset.

	Turing Machine Bytecode				
	unoptimized		optimized		
	#	size	#	size	% saved
Rules	4	32 B	4	32 B	0 %
Conditions	11	66 B	7	42 B	36.4 %
Statements	10	80 B	10	80 B	0 %
Slots	26	208 B	21	168 B	19.2 %
Expressions	38	228 B	20	120 B	47.4 %
Initializers	7	28 B	5	20 B	28.6 %
Variables	35	140 B	17	68 B	51.4 %
Facts	13	260 B	13	260 B	0 %
Properties	56	336 B	56	336 B	0 %
Total		**1,392 B**		**1,140 B**	**18,1 %**

Table 5.3: Savings achieved by bytecode optimization of the Turing Machine ruleset.

	Generic Role Assignment Bytecode				
	unoptimized		optimized		
	#	size	#	size	% saved
Rules	14	112 B	14	112 B	0 %
Conditions	101	606 B	63	378 B	37.6 %
Statements	39	312 B	35	280 B	10.3 %
Slots	186	1,488 B	64	512 B	65.6 %
Expressions	235	1,410 B	61	366 B	74.0 %
Initializers	3	12 B	2	8 B	33.3 %
Variables	218	872 B	51	204 B	76.6 %
Facts	2	40 B	2	40 B	0 %
Properties	6	36 B	6	36 B	0 %
Total		**4,902 B**		**1,950 B**	**60,2 %**

Table 5.4: Savings achieved by bytecode optimization of the Generic Role Assignment ruleset.

Examining the optimization in detail, we find that the savings are to be attributed mostly to redundant expressions, variables and, to a lesser degree, slots. In fact, these three constructs are heavily used when implementing rulesets. When looking at usage patterns however, it becomes obvious that there are only very few different instances of these elements, which are repeated very often. Taking the complete definition of the Coverage ruleset as listed in Section B.1 for example, there are in fact 107 expressions to be found in the code. But on closer inspection it turns out that only 24 of these are unique. The other 77.6 % are duplicates and as such discarted when optimizing the bytecode.

In contrast, rules, facts and properties have not been optimized at all. This also makes

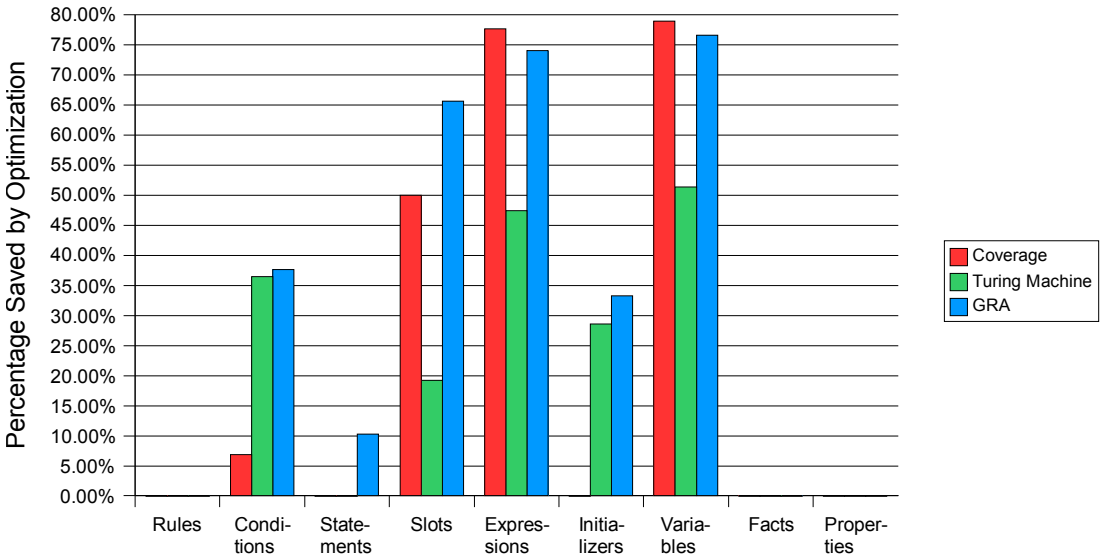

Figure 5.8: Percentage saved against type of data structure.

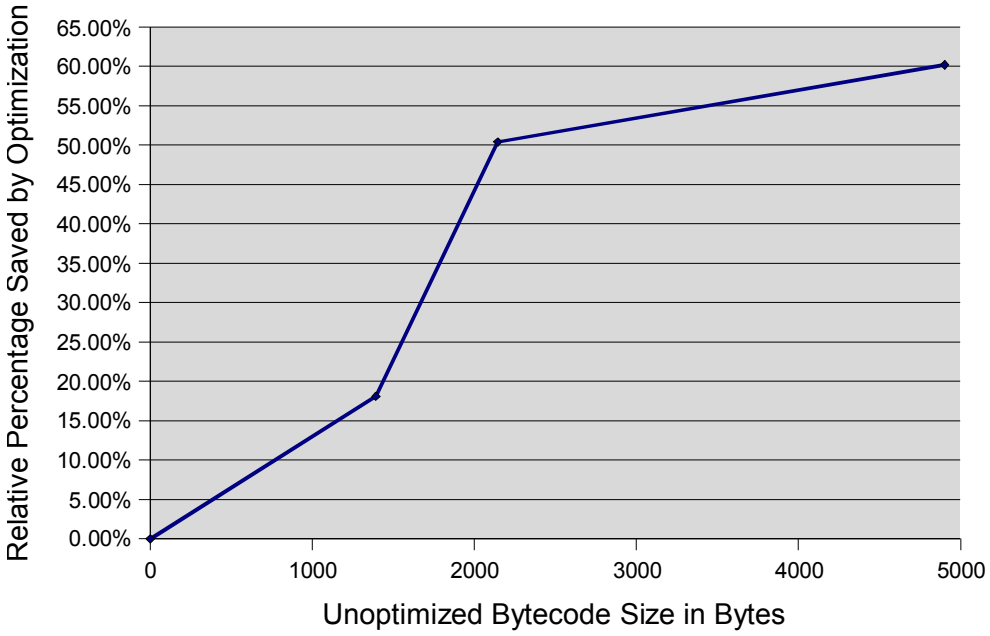

Figure 5.9: Size of unoptimized bytecode against percentage saved.

sense as there are no redundant rules in any of the rulesets examined. Facts and properties cannot be optimized at all even if they were identical, because, contrary to other elements of a ruleset, they may be modified at runtime and hence even properties that are identical at compile-time may be assigned different values at runtime and hence need separate memory to store them.

Interestingly and as shown in Figure 5.9, the relative percentage saved by optimization increases with the unoptimized size of the bytecode image. Only evaluating three rulesets does certainly not provide enough sample points to draw strong conclusions. Still, based on the data available at the moment, it appears that bytecode size scales well with the complexity of rulesets, which we find encouraging.

5.2 Prototype Implementation: FACTS-hs

The FACTS middleware architecture as proposed in Chapter 4 was first implemented as purely functional prototype in the Haskell programming language. This section explains the reasons why this somewhat unusual approach was taken, gives an overview of the implementation and presents relevant code fragments of the FACTS-hs prototype.

5.2.1 Rationale

As [Tho99, p. 449] points out, a functional design or a prototype can be most useful even if the ultimate goal is an imperative solution. In our case the advantages were as follows:

- Initially, the basic concepts were not well understood beyond traditional expert system and their exact semantics changed while new requirements and interdependencies were discovered. During this phase of rapid prototyping, the emphasis on concise functional definitions helped the project to stay coherent.
- After finishing the implementation of FACTS-hs, the definitions of the functions serve as formal specification of the system. Further, the definitions of the data types were used as basis for `lex` and `yacc` definition for the frontend of the FACTS-rc rulesets compiler. Based on these we were able to automatically generate the grammar of the ruleset definition language as given in Appendix A.
- Special cases in the internal workings of the rule engine have already been identified and implemented successfully in the prototype. The knowledge gained can be transferred easily to the imperative implementation.
- Higher order function and their capability of using functions as parameters makes the code base very compact and easy to maintain, while at the same time preserving type

safety.

The availability of the functional prototype allowed us to run test cases and check the semantics of the system very early in the development process, thereby improving overall quality.

5.2.2 Overview

The core of the FACTS-hs system is implemented as a Haskell module. Its public interface contains constructors for the condition and statement primitives and functions to create rule, fact and function entities as well as slots and rulesets. For testing purposes there are also functions that construct a sensor node, a sensor network and based on these run a simple simulation.

Following the functional paradigm, the simulation runs by iteratively transforming the current state of the sensor network – including all nodes and their respective rules and fact repositories – into the subsequent state. For all nodes the conditions of their local rules are checked against the fact repositories and the statements are executed if the given facts suffice for the rule to fire. In order to implement unique IDs of facts the simulation environment provides a global time that is incremented whenever a fact is modified or after a complete run of the simulated rule engines on all sensor nodes has been completed. The current notion of time of the sensor nodes is known at the beginning of each simulation step which allows for the simulation to supports the injection of facts into the fact repositories of one specific are all sensor nodes. As external events appear to the rule engines as new facts in their repositories, the injection method can be used to simulate sensor readings at certain points in time during the simulated deployment of the sensor network.

5.2.3 Relevant Code Fragments

Listing 5.5 is a shortened version of the main "loop" of the functional simulation. A simulation step is broken down into several operations: In lines 8 and 9 the events for the current simulation step are filtered out of the global event list. Line 7 processes these events by updating the state of the network accordingly. It then proceeds to use this new state as input for the central processing of the network. Together with the calculation of the time for the next simulation step in line 4 this concludes the processing of the current state.

Given a lists of events, the `processState` function can be reapplied to its own output. Repeating this process using Haskell's built-in `iterate` function results in an infinite list of states which holds the state of the simulated sensor network at any given time.

Listing 5.6 shows the complete logic that decides whether to fire a rule or not and updates the simulated state of the sensor network accordingly. Taking the MAC address as ID of

Listing 5.5: Functional simulation main loop.

```
1 processState :: [Event] -> State -> State
2 processState events state =
3   (State (step + 1) nextStepTime abortTime newNetwork newRxQueue)
4   where nextStepTime = time + simulationStepTime
5         (State step time abortTime _ _) = state
6         (State _ newTime _ newNetwork newRxQueue) =
7             processNetwork (processEvents state currentEvents)
8         currentEvents =
9             filter (\(Event eventStep _ _) -> (eventStep == step)) events
```

Listing 5.6: Functional rule firing logic.

```
1 applyRule :: State -> MAC -> Rule -> State
2 applyRule state mac (Rule identifier _ conditions statements)
3   | oneFactIsModified && allConditionsAreTrue =
4       foldl (\state -> applyStatement state mac) state statements
5   | otherwise = state
6   where oneFactIsModified = or (map (containsModifiedFact facts) conditions)
7         allConditionsAreTrue =
8             and (map (evaluatesAsTrue facts Nothing) conditions)
9         facts = getFacts state mac
```

the current sensor node and the rule to be applied to the fact repository of said node as additional parameters, this function can be broken down into the following operations: As stated in line 3, a rule fires only if at least one fact is tagged as modified and all conditions of the rule evaluate as true. If this is the case, a rule is applied by folding its statements into the current state of the simulation in line 4, Otherwise the state is returned unchanged in line 5 as the rule did not fire. The calculations whether a fact referenced in the conditions is modified and whether all conditions are true are given in lines 6 and 7 respectively.

Just like this function definition, several other code fragments were reused conceptually when implementing the FACTS-re runtime environment in C to run on the ScatterWeb platform, which is the topic of the following section.

5.3 Implementation on ScatterWeb: FACTS-re

The implementation of the FACTS-re runtime environment on the ScatterWeb platform is still work in progress as of this writing. We are confident that it can be implemented without major difficulties as similar bytecode interpreters have already been completed successfully in [WLT] and [Pie05]. Additionally, the functional implementation in FACTS-hs defines exact semantics of the middleware. Re-implementing them in an imperative programming language is perfectly feasible.

In the following, we will summarize the current state of the FACTS-re implementation.

5.3.1 Overview

FACTS-re is implemented as a ScatterWeb user application that runs on top of the C API provided by the ScatterWeb firmware. As already hinted at in Section 5.1.3, the EEPROM of the ScatterWeb sensor node is used to store the bytecode as well as the fact repository. Bytecode data items are retrieved from the EEPROM only when required, thereby keeping RAM usage to a minimum. This comes at the price of a rather slow execution speed. In the future we plan to improve this situation by using RAM more aggressively for caching of frequently used bytecode items. The challenge will be to dynamically adjust the number of items cached for performance while still retaining enough memory for the rule engine to operate robustly.

To integrate FACTS-re with the ScatterWeb firmware, we have created a new packet type to encapsulate facts as they are being sent from one fact repository to another. We plan for the receiving end to support both filtering of received facts as well as adding them directly to the repository. The first behavior is desirable if the application ruleset only deals with facts sent specifically to the sensor node it is located on, and all other facts are to be discarded upon reception. The second option allows for rulesets to implement routing algorithms. Filtering received facts and possibly discarding them will be the default behavior, however routing rulesets may switch to processing new facts themselves at any time by issuing a function call to the firmware.

Additionally, several hocks into the firmware are in place that will allow for even more function calls to interact with the hardware, including the toggling of LEDs for debugging.

5.3.2 Relevant Code Fragments

In Listing 5.7 we show the main loop of the rule engine as implemented using the ScatterWeb firmware API.

It loops over the addresses of all rules in the bytecode, and retrieves the current rule from the EEPROM in line 6 using the `IO_read()` function provided by the firmware. As compared to Listing 5.6, both requirements on whether a rule should fire or not are implemented in one function call in line 7, thereby avoiding repeated looping over the fact repository. If a rule fires, its statements are executed in line 8.

Also note how the structure of this loop resembles the one proposed in Listing 5.3. The care taken in the layout of the bytecode now pays off, and we use as little memory as possible.

Listing 5.7: Imperative rule firing logic.

```
1  bool FACTS_run() {
2    UINT16 rule_addr;
3    rule_t rule;
4
5    for(rule_addr = header.rules_first; rule_addr <= header.rules_last; rule_addr
         += sizeof(rule_t)){
6      IO_read(rule_addr, (UINT8*) &rule, sizeof(rule_t));
7      if(checkConditions(rule.conditions_first, rule.conditions_last))
8        applyStatements(rule.statements_first, rule.statements_last);
9    }
10 }
```

We are currently developing FACTS-re using the ScatterWeb on `ns-2` compatibility layer that will be the topic of the next chapter.

Chapter 6

Simulation: ScatterWeb on `ns-2`

Rather than part of the main FACTS project, the ScatterWeb on `ns-2` effort is a byproduct that serves as a development tool. As we anticipated the development of a distributed application such as a middleware architecture to be rather complex, we wanted to avoid having to implement the FACTS-re runtime environment directly on the actual ScatterWeb sensor nodes. Instead we chose to implement a compatibility layer that allows execution of ScatterWeb user applications as part of the `ns-2` network simulator.

After a detailed overview of goals and difficulties in Section 6.1, we will discuss different approaches to solve the problems in Section 6.2. We describe the details of our implementation in Section 6.3 and evaluate these results in Section 6.4.

6.1 Overview

The goal of this subproject is to run ScatterWeb user applications with as little modifications as possible on `ns-2` while retaining the semantics of the real ScatterWeb sensor nodes. This is to be achieved by implementing a layer of glue code between the ScatterWeb user application and `ns-2` that implements the C API of the ScatterWeb firmware as shown in Figure 6.1.

The task at hand is not trivial. There are several constraints that make the implementation of the layer of glue code challenging:

- The core and most components of `ns-2` are implemented in C++, while ScatterWeb user applications are written in C, against a C API, and otherwise closely tied to the actual hardware.
- We need to integrate the header files of the real ScatterWeb firmware because they define data types and constants which are used by the user applications.
- All components of `ns-2` are statically linked into the `ns` binary at compilation time. We need to link the ScatterWeb user application object code into the binary, possibly

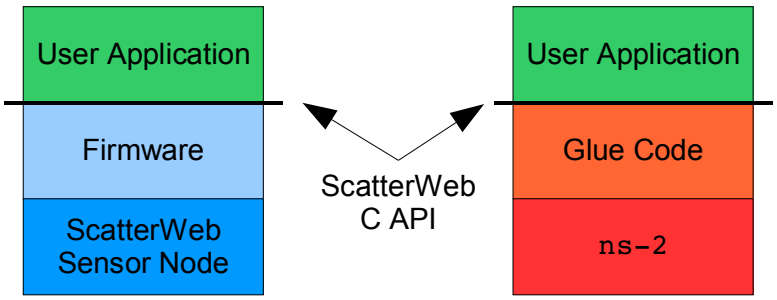

Figure 6.1: Conceptual sketch of running ScatterWeb on ns-2 by reimplementing the ScatterWeb C API.

dealing with clashing symbols.

- ScatterWeb user applications are written to run on a dedicated embedded processor. When simulated by ns-2, several user applications suddenly find themselves running as part of the same process. Hence the global variables need to be replicated for each running instance of the user application.
- The layer of glue code and with it ScatterWeb user applications need to be integrated into the build system of ns-2.

While trying to meet these constraints, it is important not to forget that modifications of the code of ScatterWeb user applications are to be avoided whenever possible. We cannot expect application developers to port their code back and forth between the real and the simulated ScatterWeb platform.

6.2 Possible Approaches

Let us carefully consider the options that we have and whether they are feasible or not:

Simple Compatibility Layer: We could write a simple compatibility layer on top of ns-2 that implements the C API of the ScatterWeb firmware. With this layer, ScatterWeb user applications could link directly against the ns binary.

This approach is not feasible, because user applications need to store state for each simulated sensor node separately. If the object code of the user application was linked only once, then all global variables would be shared between all simulated sensor nodes, or, to use object-oriented terminology, the global variables would be static. Hence we need to provide the ScatterWeb user application with the semantics of true objects that can be created dynamically with their own variables at runtime.

Automatic Conversion from C to C++: As the C code of ScatterWeb user applications needs to be integrated into C++ objects, it makes sense to look for tools to take care of this conversion automatically. `ctocpp` is one of these tools and already has been introduced in Section 3.4.

Unfortunately, the experience with automatic code conversion has been frustrating. `ctocpp` did not handle inclusion of header files properly, nor does it generate code for member variables, or, based on these, constructors and destructors. To make things worse, it does not parse compiler directives as used in the ScatterWeb source. Consequently, we decided that automatic code conversion would not be the way to go.

As approaches that are either commonly used or elegant seem to fail to meet the requirements, we decided to implement the glue code layer as a custom hack that gets the job done but is neither commonly used (to our knowledge) nor elegant. In order to justify this approach in light of the requirement of maintainability from the software engineering point of view, we have tried to tightly localize the ugly aspects of the glue code layer in a few files and document exactly how we proceeded. The next section will explain our approach in detail.

6.3 Implementation

We have implemented the layer of glue code between `ns-2` and the ScatterWeb user application as a set of two C++ classes.[1] The `ScatterWebAgent` class reimplements the ScatterWeb firmware API in C++ and takes care of the interaction with `ns-2`. The `ScatterWebUserAppAgent` inherits this functionality from `ScatterWebAgent` and additionally pulls in the C code of the user application that is to be run on the simulated sensor node. In other words, `ScatterWebAgent` takes care of the functionality of the firmware, while `ScatterWebUserAppAgent` constructs the logical unit of firmware and user application that is implicitly created when deploying firmware and user application object code on one sensor node. To be more precise, there must be one specialized `ScatterWebUserAppAgent` per specific user application, i.e. one needs to slightly adapt `ScatterWebUserAppAgent` for each user application. However, the changes typically affect only very few lines of code and are well documented.

Being derived from the `Agent` superclass, a `ScatterWebAgent` is integrated as Source / Sink into the simulated `ns-2` network stack as shown in Figure 6.2. As this subproject

[1] Actually, there are three C++ classes, but only two matter from the conceptual point of view. The third provides functionality to make the simulation more realistic later on.

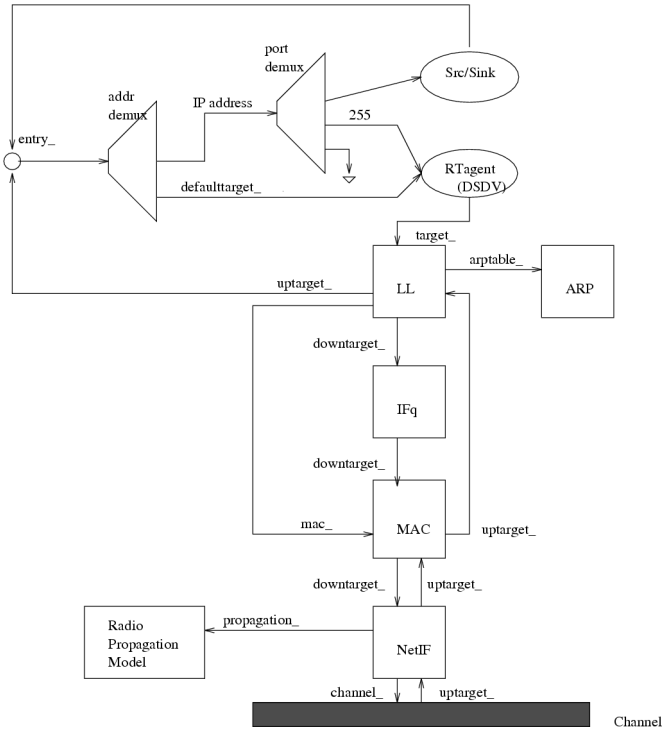

Figure 6.2: Schematic of a ns-2 mobilenode. *(Image taken from [ns005])*

is focused on implementing a proof of concept rather than a realistic simulation, we set the other components of the stack to sensible defaults, thereby enabling communication between the simulated ScatterWeb sensor nodes. Specifically, we use the `TwoRayGround` radio-propagation model, the `OmniAntenna` antenna model, the `WirelessPhy` network interface type, the `Simple` MAC layer, the `DropTail/PriQueue` interface queue, and the `DumbAgent` for routing. These and other parameters will have to be refined in the future in order to achieve a more realistic simulation.

Figure 6.3 illustrates the interaction between `ns-2`, `ScatterWebAgent`, `ScatterWebUserAppAgent`, and the user application at runtime. For the example of receiving and sending a packet, the diagram illustrates how `ScatterWebAgent` translates a call to its `recv()` (which is part of the interface of a `ns-2` agent) to the appropriate callback of the ScatterWeb user application. When sending a packet, the user application calls the `Net_send()` method of the `ScatterWebAgent` object, which it translates into the `send()` function of the `ns-2` simulation.

The diagram also shows a third C++ class, `ScatterWebAgentTimer`, which interacts closely with `ScatterWebAgent`. Its purpose is to schedule periodic events for the simulated firmware such as the interrupt-driven timer tick on the real sensor nodes.

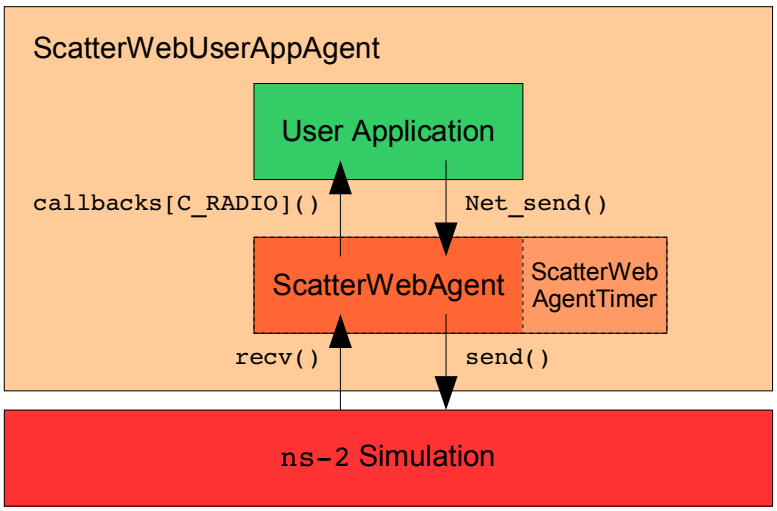

Figure 6.3: Interactions within the glue code layer at runtime.

6.3.1 Linking C Code into ns-2

The most difficult part of our approach is the `Net_send()` method: While implemented as method of the `ScatterWebAgent` class, it must look just as a C style function from the point of view of the user application code. Further, as there may be multiple `ScatterWebAgent` objects at runtime, the user application needs to call the method on the correct object.

To achieve this goal, we misuse the C preprocessor (`cpp`): Listing 6.2 shows a shortened version of the `ScatterWebUserAppAgent` class definition. In lines 3 to 9 it first defines its public interface consisting of the constructor and the `Process_init()` method, which conceptually belongs to the user application. The crucial lines are however lines 15 to 18. In line 15 we include a list of `defines` which map the C style API functions as expected by the ScatterWeb user application to the methods implemented by `ScatterWebAgent`. For example, the conversion of the `Net_send()` function looks as follows:

Listing 6.1: Mapping C API functions to C++ methods.

```
1 #define Net_send ScatterWebAgent::instance->Net_send
```

There are similar `defines` for all 114 functions of the ScatterWeb firmware API. As the API is fairly stable, this list of `defines` only needs to be created once and can be reused for all user applications afterwards.

In line 16 of Listing 6.2 we add another `define` for the `Process_init()` function of the user application. We cannot do this together with the `defines` for the other functions, as we need to know the name of application wrapper class, which in this case is

Listing 6.2: Definition of the ScatterWebUserAppAgent wrapper class.

```
1  #include "../../../../Simulation/src/ScatterWebAgent.h"
2
3  namespace ScatterWeb {
4    class ScatterWebUserAppAgent : public ScatterWebAgent {
5    public:
6      ScatterWebUserAppAgent();
7      void Process_init();
8    };
9  }
10
11 [...]
12
13 ScatterWebUserAppAgent::ScatterWebUserAppAgent() : ScatterWebAgent() {}
14
15 #include "../../../../Simulation/src/ScatterWebFirmwareWrapper.h"
16 #define Process_init ScatterWebUserAppAgent::Process_init
17 #include "ScatterWeb.Event.c"
18 #include "ScatterWeb.Process.c"
```

"ScatterWebUserAppAgent". Invocation of Process_init() will happen from the super class via polymorphism. Finally in lines 17 and 18, we include the C code of the user application into the definition of the C++ wrapper class.

Admittedly, lines 15 to 18 use language features in unusual ways and integrating the ScatterWeb user application into a C++ framework is more a hack than an elegant solution. However, the import code boils down to merely four lines and is independent from the code of the user application. Further, it will not require frequent updates and it is well documented. In this light, we think it is an acceptable solution for the problem at hand.

6.3.2 Connecting the Network Stack

Now that we have established the means of interaction between the ns-2 simulator and the ScatterWeb user application, let's have a look at the details of this interaction taking place: A shortened version of the Net_send() method of the ScatterWebAgent class is given in Listing 6.3. This method is called by the ScatterWeb user application or the simulated firmware in order to send a packet over the network. After calculating the sequence number of the ScatterWeb packet in line 4, the method allocates a ns-2 packet and its payload object in lines 5 and 6. The ScatterWeb packet is then copied into the the payload of the ns-2 packet in lines 7 and 8, before line 9 hands the packet down to the lower layers of the ns-2 simulation.

Listing 6.4 shows the recv() method of the ScatterWebAgent class, which is invoked by the ns-2 simulation when a packet has been received by the lower networking layers of

Listing 6.3: Glue code for sending a packet.

```
1  bool ScatterWebAgent::Net_send(packet_t* packet, fp_char_t callback) {
2    [...]
3
4    packet->num = txNum++;
5    Packet* pkt = allocpkt();
6    PacketData* data = new PacketData(sizeof(packet_t));
7    memcpy(data->data(), packet, sizeof(packet_t));
8    pkt->setdata(data);
9    send(pkt, 0);
10
11   [...]
12   return True;
13 }
```

Listing 6.4: Glue code for receiving a packet.

```
1  void ScatterWebAgent::recv(Packet* pkt, Handler* handler) {
2    instance = this;
3    tcl = &Tcl::instance();
4    memcpy(&rxPacket, pkt->accessdata(), sizeof(packet_t));
5    rxState = RXSTATE_FULL;
6    Packet::free(pkt);
7    runModule |= MF_RADIO_RX;
8    loop();
9  }
```

the simulated sensor node. In line 2 the `instance` variable is set to point to the current object. `instance` is a static field of the `ScatterWebAgent` class, that gives the C code of the user application a reference to the object it is part of. Therefore `instance` needs to be adjusted whenever the lower layers of the `ns-2` simulation transfer control to a `ScatterWebAgent` object. Line 3 is generic `ns-2` I/O setup code. In lines 4 to 8 the code processes the packet that was just received: It copies the payload of the `ns-2` packet into the ScatterWeb packet buffer and frees the packet. In lines 5 and 7 the state of the sensor node is updated to just having received a full packet. Finally in line 8, the main loop of the ScatterWeb firmware is executed once, just as if an interrupt had occurred on the real sensor node.

6.3.3 Simulating Timer Interrupts

One element still missing to a complete simulation of a ScatterWeb sensor node is the notion of all sensor nodes executing their respective applications concurrently. On the real ScatterWeb platform, the hardware provides timer interrupts that allow periodic tasks to be scheduled. While it is theoretically possible to use `ns-2` timers to simulate each timer

Listing 6.5: Glue code to simulate timer interrupts.

```
1  void ScatterWebAgent::loop() {
2
3    [...]
4
5    if(numberTimers > 0) {
6      runModule |= MF_TIMER;
7      _ticks = timers[0].ticks;
8      timer->resched(_ticks / SCATTERWEB_TICKS_PER_SECOND);
9    }
10 }
```

interrupt that occurs on a sensor node, the resulting overhead is prohibitively expensive for a simulation that needs to scale up to possibly several hundred simulated sensor nodes.

In order to run the simulation as efficiently as possible, we only simulate those timer interrupts that the simulated sensor node is actually waiting for: Listing 6.5 shows a small excerpt from the main loop of the simulated sensor node. Line 5 checks whether there are any active timers registered in the firmware. If this is the case, line 7 retrieves the time at which the next timer is set to expire and line 8 reschedules the simulation of the sensor node until said time. Once the simulation of this sensor node is resumed, the state of the firmware needs to hold the information that it is running in response to a timer interrupt and hence line 6 sets the global flags accordingly.

6.3.4 Minor Fixes

By now, all major components are in place. However there are few rough edges that still need to be taken care of:

- As already seen in line 3 of Listing 6.2, C++ namespaces are used. This is done in order to allow for duplicate type definitions, e.g. both `ns-2` and ScatterWeb define a `packet_t` type and both types need to coexist in the simulation.
- In the ScatterWeb firmware headers, we used `ifdefs` to redefine certain data types for language compatibility, e.g. C++ has a predefined data type `bool`, while C has not.
- Also in the ScatterWeb firmware headers, we added `ifdefs` around `includes` of hardware-specific header files.

With these minor fixes, the layer of glue code between ScatterWeb and `ns-2` is complete.

6.4 Evaluation

The simulation of ScatterWeb on the `ns-2` network simulator does work. We have successfully run several simulation setups with different ScatterWeb user applications and collected simulation traces. A screenshot of the Network Animator (`nam`) displaying one of these traces is shown in Figure 6.4.

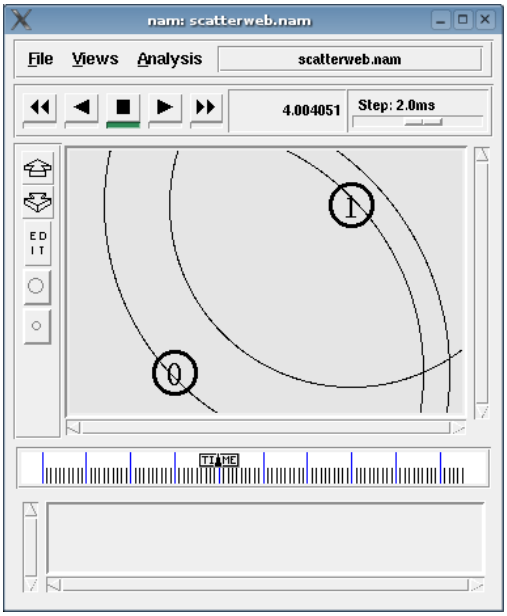

Figure 6.4: A trace of a ScatterWeb simulation shown on `nam`.

There are numerous advantages of running ScatterWeb simulations on `ns-2`: Compared to the work required for deploying a large number of sensor nodes in order to test a user application, the work of setting up a simulation is trivial. It is even possible to simulate very large sensor networks, for which it would otherwise be impossible to procure enough real sensor nodes. On a smaller scale, even the development of small applications is faster, because the development cycle of implementing, compiling and testing does not involve the time consuming act of flashing the binary images onto the sensor nodes. Perhaps even more importantly, debugging of large distributed applications, such as routing or load balancing algorithms, is quite difficult on real sensor networks. When simulating, it is not only easy to trace events in the network, one can also resort to standard debugging tools such as `gdb`. As a secondary advantage, the ScatterWeb on `ns-2` subproject uses a different compiler than the one used traditionally in ScatterWeb. As a result, several inconsistencies in the ScatterWeb source code were discovered and fixed, thus improving the overall quality of the ScatterWeb codebase.

At this stage, we have not worked on the integration of sensor readings into the simulation, leaving this for future work (see Section 8.2). Our patches are however already being included in the ScatterWeb project (see [sca]).

Chapter 7

A Use Case

Up until now, we have illustrated the concepts and workings of the FACTS middleware architecture for wireless sensor networks using either theoretical examples foreign to our application domain, such as the Turing Machine example in Section 5.1, or simplified code fragments, such as those found in Section 4.5.2. In this chapter we will underline the real-world relevance of FACTS by showing how a real-world WSN application can be implemented in FACTS.

Section 7.1 re-introduces Generic Role Assignment as the application that we will use as example for this use case. Sections 7.2 and 7.3 present our implementation, followed by an evaluation in Section 7.4.

7.1 Use Case: Generic Role Assignment

The example we have chosen is the Generic Role Assignment (GRA) framework as described in [RFMB04]. We believe GRA to be a good example because it is a state-of-the-art WSN concept proposed by an active research group in this domain, and it has only been proposed in September 2004 in [RFMB04] and implementation details will not be published until November 2005 in [FR05].[1]

To make things more interesting and as its name already suggests, GRA itself aims to be a generic framework. Preserving this level of abstraction in the FACTS implementation of GRA illustrates the power of the underlying concepts of our architecture.

Generic Role Assignment has already been introduced in Section 2.4.4, however we will quickly recapitulate the fundamental idea: GRA tackles a self-configuration problem commonly found in wireless sensor networks. Nodes have to decide which role they will perform after the network has been deployed and during its operation. Possible roles differ depending on the task of the network, and if a task requires several complementary services, a node may even have several roles, one for each service. Typical services include coverage,

[1] A copy of this paper was available on the authors homepage as of this writing.

Listing 7.1: Role definition for Coverage as formulated in the Generic Role Assignment framework. *(Code taken from [FR05])*

```
1 ON :: {
2   temp-sensor == true &&
3   battery >= threshold &&
4   count(2 hops) {
5     role == ON &&
6     dist(super.pos, pos) <= sensing-range
7   } <= 1 }
8 OFF :: else
```

clustering and in-network aggregation, as already introduced in Section 1.1.4. GRA introduces a language in which these roles can be defined and proposes a distributed algorithm that allows sensor nodes to assign roles to themselves based on their own local information and information they retrieve from neighboring nodes.

Listing 7.1 gives the GRA implementation of Coverage, a service which saves power by turning off nodes whose observations of the environment overlap with sufficient other nodes in the vicinity and are thus redundant.

As can be seen, the Coverage listing defines two states, `ON` and `OFF`, that are used to switch the sensor node either on or off, depending on whether it is equipped with a temperature sensor, has enough battery power left, and whether or not more than one other node within it's sensing range is already switched on.

The role assignment algorithm as summarized in [FR05, Section 4.1] is

> *"[...] built around local* cache tables *maintained at each node, which contain a collection of (local and remote) properties that are relevant for role assignment. Eventually, the node will refer to its cache table to assign its own role, based on the information it has learned about its neighbors up to that time."*

The cache table is also referred to as the *property directory* when the emphasis is more on the conceptual rather than the implementation side. The process of filling the property directory, and deciding which role to assume is subdivided into several phases: initialization, property propagation, and local rule evaluation. Further, the algorithm must take into account that node and network conditions may change over time, and it must ensure that the role assignment procedure terminates after a few iterations, leaving each node with a stable role.

Note that in order to maintain the flexibility of Generic Role Assignment, the role assignment algorithm must the strictly independent from the high-level role definitions. Hence we will discuss each of them separately in the following sections.

7.2 Generic Role Assignment Implemented in FACTS

The two key components of an implementation of Generic Role Assignment are the property directory to hold the properties of the sensor nodes and the algorithm to distribute these properties over the network. We have chosen to implement the property directory as a set of facts, one for each property. The distribution algorithm is implemented in the form of ten rules. The complete code is given in Listing B.3.

Note that the terminology might be slightly confusing because both FACTS and GRA make use of the term *"property"*: In FACTS, a property is a key-value-tuple belonging to one fact, in GRA a property describes a certain aspect of a physical sensor node and is stored in the property directory together with some data internal to the role assignment algorithm. Just as for all fact names, we will use a fixed-width font for the GRA `property` facts in the following.

7.2.1 Implementing the Property Directory

`property` facts are structured as shown in Figures 7.1 and 7.2. The structure is somewhat similar to the fields of the cache table as described in [FR05, Figure 2]. The `key`, `value`, and `source` fields are exact matches, except for our `source` field having a numerical value. Our `positionX` and `positionY` fields replace the field for hop counting, as given that the position of the sensor nodes is known (see Listing 7.1, line 6) it makes more sense to use a distance predicate for the `count`-operator (see Listing 7.1, line 4). Otherwise the role definition would have the implicit assumption that the sensing range is always smaller than the radio range. Finally, `timestamp`, `isUpdate`, and `needsFlushing` are fields newly introduced in our implementation, which will be explained when discussing the respective rules. Note however that for properties that are local to each sensor node and not propagated over the network, we omit the `timestamp` and `isUpdate` fields, thus saving memory.

In our GRA ruleset, we have implemented the addition of a new local property to the property directory in the form of two rules as shown in Listing 7.2. The first rule fires whenever a new interesting `property` fact is added to the local fact repository. In this context, *"interesting"* refers to those `property` facts that need to be propagated to other sensor nodes as part of the role assignment algorithm. Interesting `property` facts are identified by their `key`, which matches the one defined in the global `graControl` fact as shown in Figure 7.3. The statements of the rule then proceed to update all relevant information of the `property` fact.

Note that the last statement of the first rule sets the `needsFlushing` property to `true`. This results in the second rule, which has the lowest possible priority of 0, to flush this fact

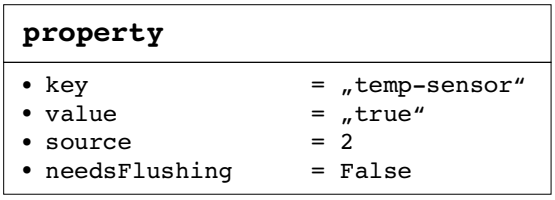

Figure 7.1: Fact representing a temperature sensor property.

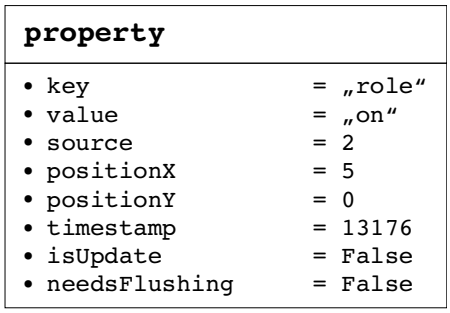

Figure 7.2: Fact representing a role property.

Listing 7.2: Addition of a new local GRA property (encapsulated in a FACTS fact) to the property directory.

```
1  rule tagMyNewProperties 200
2  <- exists {property
3    <- eval ({this key} == {graControl interestingKey})
4    <- eval ({this owner} == systemID)
5  }
6  -> set myNewPropertySource = systemID
7  -> set myNewInterestingPropertyPositionX = systemPositionX
8  -> set myNewInterestingPropertyPositionY = systemPositionY
9  -> set myNewInterestingPropertyTimestamp = myNewInterestingPropertyTime
10 -> set myNewInterestingPropertyIsUpdate = false
11 -> set myNewInterestingPropertyNeedsFlushing = true
12
13 rule flushProperties 0
14 <- eval ({property needsFlushing} == true)
15 -> set {property needsFlushing <- eval ({this modified} == true)} = false
16 -> flush {property needsFlushing <- eval ({this modified} == true)}
```

at the very end of the run of the rule engine. This interaction ensures that the facts, that were just updated and are thus internally tagged as modified, have their modified flags removed and do not cause the first rule to fire again during the next run of the rule engine.

7.2.2 Implementing Property Propagation and Role Updates

Property propagation and role updates are controlled by the `graControl` fact as shown in Figure 7.3. The `interestingKey` property of this fact describes which `property` facts should be propagated, thus implementing the notion *"that a node can decide which of its property values are relevant to what neighbors"* as stated in [FR05, Section 4]. The `newValue` property of the `graControl` fact specifies which `value` the `property` fact matching `interestingKey` should be assigned. `timeout` controls the delay until when this assignment should be made, as changing the `value` immediately might result in unstable behavior of the distributed algorithm. Setting the `timeout` property to -1 makes it inactive, i.e. no values need to be updated.

On the implementation side, it makes sense to subdivide the problem into three separate tasks:

1. Broadcast interesting `property` facts and update local values upon reception (see Listing 7.3).
2. Filter `property` facts that are not of interest either because they are out of range or duplicates (see Listing 7.4).
3. Update the timeout and change the role of the local node once the timeout expires (see Listing 7.5).

In the following we will have a detailed look at each of these steps one by one and point out peculiarities of the implementation.

graControl	
• interestingKey	= „role“
• newValue	= „on“
• timeout	= -1

Figure 7.3: GRA control fact.

Listing 7.3: GRA property propagation.

```
1  rule propagatePropertiesInSensingRange 50
2  <- exists interestingPropertyIsNotUpdate
3  -> set interestingPropertyIsNotUpdate = true
4  -> send 0 systemTxRange interestingPropertyIsUpdate
5  -> set interestingPropertyIsUpdate = false
6
7  rule replaceUpdatedProperties 150
8  <- exists {property <- eval ({this isUpdate} == true)}
9  -> define tmp [updateID = {property id
10   <- eval ({this isUpdate} == true)
11   <- eval ({this source} != systemID)
12 }]
13 -> set {tmp updateSource} = {property source <- eval ({this id} == {tmp
       updateID})}
14 -> set {tmp updateKey} = {property key <- eval ({this id} == {tmp updateID})}
15 -> retract {property
16   <- eval ({this source} == {tmp updateSource})
17   <- eval ({this key} == {tmp updateKey})
18   <- eval ({this id} != {tmp updateID})
19 }
20 -> set {property isUpdate <- eval ({this id} == {tmp updateID})} = false
21 -> retract {tmp}
```

Broadcast and Update

Listing 7.3 consists of two rules, one taking care of transmitting `property` facts and the other one of handling received `property` facts. The `propagatePropertiesInSensingRange` fires when `property` facts that are not flagged as being an update themselves become available. For these facts, the rule sets the `isUpdate` property to `true` in line 3, broadcasts them to all sensor nodes in the transmission range, and then sets the `isUpdate` property back to false. The function of the `isUpdate` property is to differentiate between locally generated and received `property` facts: Only received `property` facts are flagged as being an update. As a side note, it is worth pointing out that the two slots being used in lines 2 and 3 as opposed to 4 and 5 are different, although they both refer to the same fact. The reason for this is that the fact they are addressing via filtering conditions is changed in line 3 and hence the filtering conditions of the slot need to be adjusted.

The second rule in Listing 7.3 deals with handling the reception of `property` facts as transmitted by the previous rule. Accordingly, it fires whenever a `property` fact flagged as an update is added to the fact repository. The first three statements extract information from the new `property` fact and store it in a temporary fact. Note how the first of these statements in lines 9 to 12 only extracts the unique `id` of the `property` fact, which is then

used to easily access the other properties in lines 13 and 14. The `retract` statement in lines 15 to 19 then removes all potential duplicates from the local fact repository, duplicates being exactly those `property` facts that were received from the same source as the current `property` fact, share the same `key`, but are *not* identical – after all we only want to retract older facts but not the current one. Finally, the rule sets the `isUpdate` property of the new `property` fact to false in line 20 and retracts the temporary fact.

Filtering

The above scheme appears to be straightforward, but unfortunately there are several corner cases to take care of in which we do not want existing `property` facts to be replaced. This filtering of newly received `property` facts is implemented in the three rules in Listing 7.4. Note that each of these rules has a higher priority than the `replaceUpdatedProperties` rule in the previous listing. They are hence executed earlier and may remove undesired `property` facts before these are processed any further.

All three rules fire when new `property` facts marked as updates are added to the local fact repository. The first rule removes all those `property` facts that should not be propagated any further because they are too far away from their originating node to be of further interest. In our example, this is true when the facts are outside the sensing range of the sensor node.

The second rule, after once again defining a temporary fact for easy access to the ID of the current `property` fact, retracts the current `property` fact if it was generated at exactly the local sensor node, i.e. the local sensor node has received its own `property` fact that was retransmitted by a remote sensor node. The crucial conditions for these facts are that the `source` of the `property` fact matches the ID of the local node (line 15) and was modified by another node (line 16).

Finally, the third rule takes care of retracting outdated `property` facts for which the sensor node already has received newer information. It follows the by now well established pattern of generating a temporary fact in lines 22 to 25. In the crucial `retract` statement in lines 26 to 33, the rule retracts the current `property` fact (line 27) only if a *different* `property` fact exists which originated from the same source, has the same key, but a newer timestamp (lines 28 to 32). This construct is particularly interesting because we have two nested slots both with the same fact names. The inner slot allows to check whether a fact different from the one currently being evaluated in the outer slot exists.

Listing 7.4: Removing unneeded properties.

```
1  rule removeOutOfRangeProperties 160
2  <- exists {property <- eval ({this isUpdate} == true)}
3  -> retract {property
4    <- eval ((sqrt (
5        (({this positionX} - systemPositionX) ^ 2)
6        + (({this positionY} - systemPositionY) ^ 2)
7      )) > systemSensingRange)
8  }
9
10 rule removeOwnDuplicateProperties 160
11 <- exists {property <- eval ({this isUpdate} == true)}
12 -> define tmp [duplicateID = {property id <- eval ({this isUpdate} == true)}]
13 -> retract {property
14   <- eval ({this id} == {tmp duplicateID})
15   <- eval ({this source} == systemID)
16   <- eval ({this owner} != {system owner})
17 }
18 -> retract {tmp}
19
20 rule removeOtherDuplicateProperties 155
21 <- exists {property <- eval ({this isUpdate} == true)}
22 -> define tmp [duplicateID = {property id <- eval ({this isUpdate} == true)}]
23 -> set {tmp duplicateSource} = {property source <- eval ({this id} == {tmp
       duplicateID})}
24 -> set {tmp duplicateKey} = {property key <- eval ({this id} == {tmp
       duplicateID})}
25 -> set {tmp duplicateTimestamp} = {property timestamp <- eval ({this id} == {
       tmp duplicateID})}
26 -> retract {property
27   <- eval ({this id} == {tmp duplicateID})
28   <- exists {property
29     <- eval ({this source} == {tmp duplicateSource})
30     <- eval ({this key} == {tmp duplicateKey})
31     <- eval ({this timestamp} >= {tmp duplicateTimestamp})
32   }
33 }
34 -> retract {tmp}
```

Listing 7.5: Change role after timeout.

```
1  rule cancelTimeoutIfUnchanged 130
2  <- eval ({graControl timeout} > 0)
3  <- eval ({graControl newValue} == myRole)
4  -> set {graControl timeout} = -1
5
6  rule countdownRoleTimeout 120
7  <- eval ({graControl timeout} > 0)
8  -> set {graControl timeout} = ({graControl timeout} - 1)
9
10 rule setMyRoleAfterTimeout 120
11 <- eval ({graControl timeout} == 0)
12 -> set myRole = {graControl newValue}
13 -> set myRoleTimestamp = myRoleTime
```

Timeout and Role Update

As the final building block of the generic part of the implementation of Generic Role Assignment as FACTS rulesets, the rules in Listing 7.5 deal with updating the timeout counter and changing the current role once the timeout expires.

The implementation is straightforward: The first rule cancels the timeout (as kept in the `timeout` property of the `graControl` fact) if the `newValue` already matches the current role. This may occur in case the local sensor node was about to change into a different role, but while waiting for the timeout to expire it has received new information that causes it to return to its previous role. As the timeout has not expired yet and the first role change has yet to occur, the solution is to simply abort the timeout thus leaving the current role unchanged. To this end the `timeout` property is set to -1.

The second rule decrements the `timeout` property until it reaches 0, at which point the third rule fires.

The third rule updates the role of the local node based on the `newValue` property stored in the `graControl` fact. Additionally, it updates the timestamp of the role `property` fact. The updated role `property` fact fires the first rule in Listing 7.3, thereby propagating the changed role of the local sensor nodes to other nodes in the neighborhood.

Having implemented Generic Role Assignment in the FACTS ruleset definition language, we now proceed to implement to Coverage specific parts in the next section.

7.3 Coverage Implemented in GRA Running on FACTS

The GRA role definition is implemented in the form of rules. They bridge the gap between the available `property` facts and the properties of the `graControl` fact that control role updates.

Listing 7.6: Rule specifying whether a sensor node is turned on.

```
1  rule coverageON 100
2  <- exists {property}
3  <- exists {property
4    <- eval ({this source} == systemID)
5    <- eval ({this key} == "temp-sensor")
6    <- eval ({this value} == true)
7  }
8  <- exists {property
9    <- eval ({this source} == systemID)
10   <- eval ({this key} == "battery")
11   <- eval ({this value} >= systemBatteryThreshold)
12 }
13 <- eval ((count {property
14   <- eval ({this source} != systemID)
15   <- eval ({this key} == "role")
16   <- eval ({this value} == "on")
17 }) <= 1)
18 -> set {graControl newValue} = "on"
19 -> set {graControl timeout} = ((systemID * {system timeout-multiplier}) + {
       system timeout-offset})
```

The Coverage algorithm implemented on top of the FACTS version of Generic Role Assignment requires four rules, two of which are shown in Listings 7.6 and 7.7. The first rule specifies when a sensor node is to be turned on, the other three rules specify when a sensor node is to be turned off. Having three rules to switch a sensor node off is significantly more overhead than the one required by the GRA implementation of Coverage (see Listing 7.1). Unfortunately, there is no workaround to this drawback as the FACTS ruleset definition language does not include the semantics of an `else` keyword (compare Section 4.4.6).

The rule in Listing 7.6 closely resembles the definition of the "`ON`" role in Listing 7.1. The check for each GRA property translates directly into an `exists` condition for simple GRA conditions. For GRA conditions involving an operator, such as the `count` keyword, there is an equivalent FACTS `eval` condition. Note that each of these conditions checks whether the `property` fact originated at the local node for the first two conditions (lines 3 to 12) or at a remote node (lines 13 to 17).

Once the rule fires, it sets the `newValue` property of the `graControl` fact to "`on`" and initializes the `timeout` property. For simplicity, we base this initialization on the ID of the local sensor node rather than a random value.

The `exists` condition of the rule shown in Listing 7.7 is the direct opposite to the `exits` condition in lines 3 to 7 of Listing 7.6. Note how the comparison operator in line 6 differs. This rule sets the local sensor node to be turned off because it does *not* have a temperature sensor available.

Listing 7.7: Rule specifying whether a sensor node is turned off.

```
1 rule coverageOFF_1 100
2 <- exists {property}
3 <- exists {property
4   <- eval ({this source} == systemID)
5   <- eval ({this key} == "temp-sensor")
6   <- eval ({this value} != true)
7 }
8 -> set {graControl newValue} = "off"
9 -> set {graControl timeout} = ((systemID * {system timeout-multiplier}) + {
    system timeout-offset})
```

There are two more similar rules as part of the Coverage on GRA implementation in FACTS. These rules trigger on the respective opposites to the conditions in lines 8 to 12 and 13 to 17 of Listing 7.6.

This concludes the discussion of the implementation of Generic Role Assignment, and on top of that a Coverage algorithm, on the FACTS middleware architecture. The complete ruleset is available in Section B.3.

7.4 Evaluation

The successful implementation of Generic Role Assignment in the form of FACTS rulesets underscores the power of our middleware architecture. While not having access to the original implementation of Generic Role Assignment, it is our impression that the implementation in merely ten rules presented in this chapter is rather elegant.

On the downside, we have not been able to run the compiled Coverage / GRA bytecode as of this writing, because the implementation of the FACTS-re runtime environment on the ScatterWeb platform is still lacking some language features. We are currently working on implementing these missing features, a task which we do not expect to be particularly challenging as several implementation of somewhat similar bytecode interpreters such as [WLT] and [Pie05] have already been completed successfully.

On the other hand, we have successfully simulated a deployment of GRA running on FACTS using FACTS-hs. Given that [FR05] also merely describes an implementation of GRA in a simulation environment, we consider the results presented in this chapter as sufficient to justify our claims.

Chapter 8

Future Work and Conclusion

In this thesis we have proposed the FACTS middleware architecture as tool to ease development of applications and services for wireless sensor networks. After a short introduction, we started by discussing related work and quickly presented the tools used during our development effort (Chapters 1, 2 and 3). In Chapter 4 we then explained the fundamental concepts of the FACTS middleware architecture and introduced the ruleset definition language as a mean for implementing rule-based applications and services. Chapter 5 discussed the steps undertaken to implement our proposals and presented the FACTS-rc ruleset compiler with an emphasis on bytecode optimization, the FACTS-hs implementation of FACTS in Haskell, and the FACTS-re implementation of FACTS on the ScatterWeb platform.

Slightly deviating from the main topic of this thesis, we discussed in Chapter 6 how to run ScatterWeb user application on the `ns-2` network simulator. Our main goal in doing so was to support the development of the FACTS-re runtime environment. Back on the main track, Chapter 7 presented a detailed use case for the FACTS middleware architecture running a state-of-the-art WSN application.

Finally, in this chapter we will evaluate our work as whole, discuss directions for future research, and conclude.

8.1 Evaluation

We set out with the goal of developing a middleware architecture for wireless sensor networks that is both data-centric and event-driven. Inspired by the programming abstractions from the expert system domain, the FACTS middleware architecture was designed around facts, rules and functions. Rule-based programming is indeed being used implicitly in several middleware approaches for wireless sensor networks, as discussed in Section 2.4. To the best of our knowledge, we are the first to explicitly propose a holistic rule-based architecture in this domain, that is both generic and expressive enough to allow for other rule-based concepts to be implemented on top of it.

In our work we have paid special attention to robustness and reproducibility. Hence, we did not settle for a single implementation that implicitly defines the detailed semantics of our system, but we created two separate implementations, one with its focus on concepts and completeness and the other one targeted towards real-world deployments. Furthermore, we have used well-known standard tools where ever possible to keep the barriers of entry of our project as low as possible.

Finally, with our implementation of ScatterWeb on `ns-2`, we created a powerful development tool with applications well outside our main project.

8.2 Future Work

There are three major directions in which to proceed based on the results presented in this thesis.

On the implementation side, we first need to finish the implementation of the runtime environment. Once this is completed, the more challenging work of optimizing the algorithms begins. We expect the read and write access to the EEPROM of the sensor node that stores the bytecode as well as the fact repository to be the main performance bottleneck in the execution of the rule engine. Avoiding access to the EEPROM should result in major performance improvements, however the obvious solution to cache the respective data in RAM is made difficult by the very limited amount of RAM. Established caching strategies may provide a starting point when looking into this problem. Still, they will need to be adapted to the fact that the bytecode contains data structures of different types and sizes and with different access patterns.

On the application-level side, the next logical step is to extend the functionality of the FACTS middleware by implementing more algorithms and services commonly used in wireless sensor networks in the form of rulesets. This will lead to questions about the dependencies between rulesets and their interactions at run-time. We envision that rulesets will eventually be used as drop-in components for sensor networks that extend the capabilities of a deployed wireless sensor network while making these advantages transparently available to the application that runs on the middleware.

The steps required to reach this goal are:

1. To add support for handling dependency resolution and compiling several rulesets into one bytecode image to the FACTS-rc ruleset compiler. This includes investigating the notion of what exactly the public interface of a ruleset should be and resolving public symbols across rulesets.

2. To implement compile-time checking for cyclic interactions between rules, i.e. given that rules react to certain facts and modify other facts, can we be sure that the execution of several interdependent rules does not loop indefinitely? This and related issues have already been discussed in the past by the knowledge engineering community, and there are good starting points for our work such as [PSA92] and [Kip92].

3. To eventually allow for the transmission of new rulesets to the sensor nodes at runtime. In order for this to work properly, the rule engine must be able to rearrange the bytecode in the EEPROM, possibly updating meta information and pointers in the current bytecode.

On the simulation side, it would be desirable to achieve better integration of the ScatterWeb on `ns-2` patches with the ScatterWeb code base and implement the remaining functionality. The more interesting challenge is however to design a proper simulation of the environment as perceived by the sensors of the simulated sensor nodes. The scope of this simulation would be similar to the one proposed in [Dow04]. As discussed in Section 2.5.2, their approach has some drawbacks. [Win05] suggests that a better approach would be to have one global `ns-2` agent for each property of the environment that is to be simulated. This agent provides an interface for the simulated sensor nodes to register for very specific events, e.g. the event that the temperature in a certain area rises above a threshold value. The sensor node that has registered for this specific event is sent one notification by the environment agent only once when the event occurs. Compared to the periodic broadcast packets as described in [Dow04] this approach would have a significantly lower overhead during the simulation.

8.3 Concluding Remarks

Seeing a project grow from conceptual sketches into fully implemented components is a very satisfying experience. In about six months we have covered quite some ground and created the FACTS middleware architecture for wireless sensor networks basically from scratch, thereby closing the previously existing gap between us and established research groups in this field.

With FACTS we have created a middleware architecture that combines the advantages of the three major middleware abstractions: FACTS supports the grouping abstraction, interpretation of bytecode, and service-oriented modularity. Our three key concepts, facts, rules and functions, map naturally to the domain of wireless sensor networks. Together with the expressiveness of the ruleset definition language, they empower developers to implement applications that are both data-centric and event-driven by design.

We have already published parts of the results described in this thesis, namely the first half of Chapter 4 and the middle of part of Chapter 5, in [TWS06] together with the coverage example discussed in Section 4.5.2. In the near future we hope to follow up these publications with both a quantitative study of the FACTS middleware running on the ScatterWeb platform and a rather technical paper about the intricacies of simulating ScatterWeb on `ns-2`.

Based on the foundations developed so far and in light of the open questions and the potential of the FACTS middleware architecture, we look forward to continuing our research.

Appendix A

Ruleset Definition Language Grammar

This is the complete grammar of the FACTS ruleset definition language. It has been generated automatically using a patched version of `y2l` as introduced in Section 3.5 with the `yacc` and `lex` definition files of the FACTS-rc ruleset compiler as input. Section 4.5 explains the semantics of the language and provides some examples. Section 5.1 covers the compilation process.

ruleset	::=	"**ruleset**" identifier block_list
identifier	::=	[a−zA−Z_][a−zA−Z_\−0−9]∗
block_list	::=	block { block }
block	::=	depends
	|	named_name
	|	named_slot
	|	fact
	|	rule
depends	::=	"**depends**" identifier
named_name	::=	"**name**" identifier "=" name
named_slot	::=	"**slot**" identifier "=" slot
fact	::=	"**fact**" name property_list_opt
rule	::=	"**rule**" identifier priority condition_list statement_list
priority	::=	[0−9]+
	|	−[0−9]+
slot	::=	identifier
	|	"{" name condition_list_opt "}"
	|	"{" name key condition_list_opt "}"
condition_list_opt	::=	[condition_list]
condition_list	::=	"< −" condition [condition_list]
condition	::=	"**exists**" slot
	|	"**eval**" "(" expression comparison_operation expression ")"

```
comparison_operation ::= "=="
                       | "!="
                       | "<"
                       | ">"
                       | "<="
                       | ">="
statement_list       ::= "->" statement [ statement_list ]
statement            ::= "define" name initializer_list_opt
                       | "retract" slot
                       | "copy" slot
                       | "send" expression expression slot
                       | "set" slot "=" expression
                       | "flush" slot
                       | "call" identifier expression_list_opt
expression_list_opt  ::= [ "(" expression_list ")" ]
expression_list      ::= expression { "," expression }
expression           ::= variable
                       | "(" unary_operation expression ")"
                       | "(" expression binary_operation expression ")"
unary_operation      ::= "count"
                       | "sum"
                       | "product"
                       | "min"
                       | "max"
                       | "sqrt"
binary_operation     ::= "+"
                       | "-"
                       | "*"
                       | "/"
                       | "^"
property_list_opt    ::= [ "[" property_list "]" ]
property_list        ::= property { "," property }
property             ::= key "=" variable
key                  ::= [a-zA-Z_][a-zA-Z_\-0-9]*
variable             ::= "true"
                       | "false"
                       | [0-9]+
```

```
                        |   -[0-9]+
                        |   quoted_string
                        |   slot
initializer_list_opt    ::= [ "[" initializer_list "]" ]
initializer_list        ::= initializer { "," initializer }
initializer             ::= key "=" expression
name                    ::= identifier
                        |   quoted_string
quoted_string           ::= "" ( "" | [a-zA-Z_][a-zA-Z_\-0-9]* "" )
```

Appendix B

Example Rulesets

B.1 Coverage Ruleset

Listing B.1: Coverage ruleset.

```
/**
 *
 * Problem: Coverage
 * Shutdown nodes that supply values for regions already covered
 *
 **/

ruleset Coverage



/**
 * Generic node setup.
 **/

name system = "system"
fact system [broadcastMAC = 0, txPower = 15]

name node = "node"

slot nodeID = {node owner}
slot nodePosX = {node posX}
slot nodePosY = {node posY}



/**
 * Coverage
 **/
```

```
30
31 fact node [state = "UNDEF"]
32
33 name init = "init"
34
35 slot coveredNodeID = {"covered" owner}
36
37
38
39 // Calculate pessimistic range and send it to neighbors.
40
41 rule range 50
42 <- exists {init}
43 -> retract {init}
44 -> set {"node" state} = "ON"
45 -> define "range" [xMin = (nodePosX - 10), xMax = (nodePosX + 10), yMin = (
       nodePosY - 10), yMax = (nodePosY + 10)]
46 -> send {system broadcastMAC} {system txPower} {"range"}
47
48
49
50 // If the node is currently turned on, check all received range facts if they
51 // cover some of the current node's regions, but don't create duplicates.
52
53 slot rangeXMaxYMaxOwner = {"range" owner
54   <- eval ({this owner} != nodeID)
55   <- eval ({this xMin} <= nodePosX)
56   <- eval ({this yMin} <= nodePosY)
57   <- eval ({this xMin} >= (nodePosX - 10))
58   <- eval ({this yMin} >= (nodePosY - 10))
59 }
60
61 slot rangeXMaxYMinOwner = {"range" owner
62   <- eval ({this owner} != nodeID)
63   <- eval ({this xMin} <= nodePosX)
64   <- eval ({this yMax} >= nodePosY)
65   <- eval ({this xMin} >= (nodePosX - 10))
66   <- eval ({this yMax} <= (nodePosY + 10))
67 }
68
69 slot rangeXMinYMinOwner = {"range" owner
70   <- eval ({this owner} != nodeID)
71   <- eval ({this xMax} >= nodePosX)
72   <- eval ({this yMax} >= nodePosY)
73   <- eval ({this xMax} <= (nodePosX + 10))
```

```
  <- eval ({this yMax} <= (nodePosY + 10))
}

slot rangeXMinYMaxOwner = {"range" owner
  <- eval ({this owner} != nodeID)
  <- eval ({this xMax} >= nodePosX)
  <- eval ({this yMin} <= nodePosY)
  <- eval ({this xMax} <= (nodePosX + 10))
  <- eval ({this yMin} >= (nodePosY - 10))
}

// first: xMax/yMax
rule xMaxYMaxCovered 30
<- eval ({node state} == "ON")
<- exists rangeXMaxYMaxOwner
-> retract {"coveredXMaxYMax"
  <- eval ({this byNode} == rangeXMaxYMaxOwner)
}
-> define "coveredXMaxYMax" [byNode = rangeXMaxYMaxOwner]

// second: xMax/yMin
rule xMaxYMinCovered 30
<- eval ({node state} == "ON")
<- exists rangeXMaxYMinOwner
-> retract {"coveredXMaxYMin"
  <- eval ({this byNode} == rangeXMaxYMinOwner)
}
-> define "coveredXMaxYMin" [byNode = rangeXMaxYMinOwner]

// third: xMin/yMin
rule xMinYMinCovered 30
<- eval ({node state} == "ON")
<- exists rangeXMinYMinOwner
-> retract {"coveredXMinYMin"
  <- eval ({this byNode} == rangeXMinYMinOwner)
}
-> define "coveredXMinYMin" [byNode = rangeXMinYMinOwner]

// fourth: xMin/yMax
rule xMinYMaxCovered 30
<- eval ({node state} == "ON")
<- exists rangeXMinYMaxOwner
-> retract {"coveredXMinYMax"
  <- eval ({this byNode} == rangeXMinYMaxOwner)
}
```

```
119 -> define "coveredXMinYMax" [byNode = rangeXMinYMaxOwner]
120
121
122
123 // If all parts are covered, the node sends a message to its neighbors that it
124 // is going to switch itself off so they retract the range fact of this node.
125
126 rule determineCoverage 40
127 <- exists {"coveredXMaxYMax"}
128 <- exists {"coveredXMaxYMin"}
129 <- exists {"coveredXMinYMin"}
130 <- exists {"coveredXMinYMax"}
131 -> define "covered"
132 -> send {system broadcastMAC} {system txPower} {"covered"}
133 -> set {node state} = "OFF"
134
135
136
137 // If a node gets a covered fact from another node, this implies that the node
138 // in question has turned itself off. Hence we need retract the range fact and
139 // possibly the coveredXY fact. Further, the current node remains switched on.
140
141 rule check 70
142 <- eval (coveredNodeID != nodeID)
143 -> retract {"range"
144   <- eval ({this owner} == coveredNodeID)
145 }
146 -> retract {"coveredXMaxYMax"
147   <- eval ({this byNode} == coveredNodeID)
148 }
149 -> retract {"coveredXMaxYMin"
150   <- eval ({this byNode} == coveredNodeID)
151 }
152 -> retract {"coveredXMinYMin"
153   <- eval ({this byNode} == coveredNodeID)
154 }
155 -> retract {"coveredXMinYMax"
156   <- eval ({this byNode} == coveredNodeID)
157 }
158 -> retract {"covered"}
159 -> set {node state} = "ON"
```

B.2 Turing Machine Ruleset

Listing B.2: Turing Machine ruleset.

```
/**
 *
 * This is the Turing Machine ruleset which implements a complete Turing
 * Machine to show the power of the rule engine.
 *
 **/

ruleset TuringMachine

name constants = "constants"
name run = "run"
name function = "function"
name tape = "tape"

// These constants are to be used by external scripts that use this ruleset.
fact constants [errorState = -1, blankSymbol = "_", right = 1, left = -1,
    neutral = 0]

// These slots make the programming below more readable.
slot constErrorState = {constants errorState}
slot constBlankSymbol = {constants blankSymbol}
slot constRight = {constants right}
slot constLeft = {constants left}
slot constNeutral = {constants neutral}

slot runState = {run state}
slot runPosition = {run position}
slot runLeftBorder = {run leftBorder}
slot runRightBorder = {run rightBorder}

slot currentTapeSymbol = {tape symbol
  <- eval ({this position} == runPosition)
}

// Note how the slots for next state, symbol, and movement filter the fact base
// dynamically based on the current state of the Turing Machine.
slot functionNextState = {function nextState
  <- eval ({this state} == runState)
  <- eval ({this symbol} == currentTapeSymbol)
}
slot functionNextSymbol = {function nextSymbol
  <- eval ({this state} == runState)
```

```
42    <- eval ({this symbol} == currentTapeSymbol)
43  }
44  slot functionNextMovement = {function nextMovement
45    <- eval ({this state} == runState)
46    <- eval ({this symbol} == currentTapeSymbol)
47  }
48
49  /*
50   * This rule deals with error handling and turns the machine off when an error
51   * is encountered.
52   */
53  rule error 200
54  <- eval (runState == constErrorState)
55  -> retract {run}
56
57  /*
58   * These two rules create the impression of an infinite tape by extending the
59   * tape with new tape facts whenever the head reaches a side.
60   */
61  rule leftBorder 150
62  <- eval (runPosition < runLeftBorder)
63  -> define tape [position = runPosition, symbol = constBlankSymbol]
64  -> set runLeftBorder = (runLeftBorder - 1)
65
66  rule rightBorder 150
67  <- eval (runPosition > runRightBorder)
68  -> define tape [position = runPosition, symbol = constBlankSymbol]
69  -> set runRightBorder = (runRightBorder + 1)
70
71  /*
72   * This rule does the main processing of the Turing Machine: As long as the
73   * machine is running, move the current state, position and symbol to a
74   * temporary fact, then update the tape fact, and the state and position
75   * attributes of the run fact.
76   */
77  rule step 100
78  <- exists {run}
79  -> define "next" [state = functionNextState, symbol = functionNextSymbol,
        movement = functionNextMovement]
80  -> set runState = {"next" state}
81  -> set currentTapeSymbol = {"next" symbol}
82  -> set runPosition = (runPosition + {"next" movement})
83  -> retract {"next"}
84
85
```

```
86
87  /**
88   * These facts implement a Turing Machine that increments a binary-coded
89   * number on the tape and then stops.
90   *
91   * See Uwe Schoening "Theoretische Informatik -- kurzgefasst" p. 83
92   **/
93
94  // These facts define the function of the Turing Machine, matching current
95  // state and symbol to the next symbol, state, and tape movement.
96  fact function [state = 0, symbol = "s0",    nextState = 0,      nextSymbol = "
        s0",    nextMovement = constRight]
97  fact function [state = 0, symbol = "s1",    nextState = 0,      nextSymbol = "
        s1",    nextMovement = constRight]
98  fact function [state = 0, symbol = constBlankSymbol,  nextState = 1,
        nextSymbol = constBlankSymbol,  nextMovement = constLeft]
99  fact function [state = 1, symbol = "s0",    nextState = 2,      nextSymbol = "
        s1",    nextMovement = constLeft]
100 fact function [state = 1, symbol = "s1",    nextState = 1,      nextSymbol = "
        s0",    nextMovement = constLeft]
101 fact function [state = 1, symbol = constBlankSymbol,  nextState =
        constErrorState,  nextSymbol = "s1",    nextMovement = constNeutral]
102 fact function [state = 2, symbol = "s0",    nextState = 2,      nextSymbol = "
        s0",    nextMovement = constLeft]
103 fact function [state = 2, symbol = "s1",    nextState = 2,      nextSymbol = "
        s1",    nextMovement = constLeft]
104 fact function [state = 2, symbol = constBlankSymbol,  nextState =
        constErrorState,  nextSymbol = constBlankSymbol,  nextMovement = constRight]
105
106 // These facts are the intial content of the tape.
107 fact tape [position = 1, symbol = "s1"]
108 fact tape [position = 2, symbol = "s0"]
109 fact tape [position = 3, symbol = "s1"]
```

B.3 Generic Role Assignment with Coverage Ruleset

Listing B.3: Generic Role Assignment with Coverage ruleset.

```
/**
 *
 * Generic Role Assignment / Coverage
 *
 * This script implements a simple role-assignment algorithm for wireless
 * sensor networks based on Generic Role Assignment as discussed in
 * Christian Frank and Kay Roemer. Algorithms for Generic Role Assignment in
 * Wireless Sensor Networks. In Proceedings of the 3rd ACM Conference on
 * Embedded Networked Sensor Systems (SenSys), San Diego, CA, USA, November
 * 2005. http://www.vs.inf.ethz.ch/publ/papers/sensys05.roleassignment.pdf
 *
 **/

ruleset GenericRoleAssignment




/**
 * Generic system setup.
 **/

name system = "system"
slot systemID = {system owner}
slot systemPositionX = {system positionX}
slot systemPositionY = {system positionY}
slot systemTxRange = {system tx-range}




/**
 * Generic Role Assignment.
 **/

name property = "property"
name graControl = "graControl"
name tmp = "tmp"

// The timeout multiplier and offset need to allow for enough simulated time
// for the nodes to reach a consistent state.
fact system [tx-range = 10, timeout-multiplier = 3, timeout-offset = 3, battery
    -threshold = 30, sensing-range = 10]

```

```
42 slot systemBatteryThreshold = {system battery-threshold}
43 slot systemSensingRange = {system sensing-range}
44
45 slot myRole = {property value
46   <- eval ({this source} == systemID)
47   <- eval ({this key} == "role")
48 }
49 slot myRoleTime = {property time        // Read-only value generated by the
       rule engine.
50   <- eval ({this source} == systemID)
51   <- eval ({this key} == "role")
52 }
53 slot myRoleTimestamp = {property timestamp
54   <- eval ({this source} == systemID)
55   <- eval ({this key} == "role")
56 }
57
58 slot myNewPropertySource = {property source
59   <- eval ({this owner} == systemID)
60   <- eval ({this modified} == true)
61 }
62 slot myNewInterestingPropertyPositionX = {property positionX
63   <- eval ({this key} == {graControl interestingKey})
64   <- eval ({this owner} == systemID)
65   <- eval ({this modified} == true)
66 }
67 slot myNewInterestingPropertyPositionY = {property positionY
68   <- eval ({this key} == {graControl interestingKey})
69   <- eval ({this owner} == systemID)
70   <- eval ({this modified} == true)
71 }
72 slot myNewInterestingPropertyIsUpdate = {property isUpdate
73   <- eval ({this key} == {graControl interestingKey})
74   <- eval ({this owner} == systemID)
75   <- eval ({this modified} == true)
76 }
77 slot myNewInterestingPropertyTime = {property time    // Read-only value
       generated by the rule engine.
78   <- eval ({this key} == {graControl interestingKey})
79   <- eval ({this owner} == systemID)
80   <- eval ({this modified} == true)
81 }
82 slot myNewInterestingPropertyTimestamp = {property timestamp
83   <- eval ({this key} == {graControl interestingKey})
84   <- eval ({this owner} == systemID)
```

```
85    <- eval ({this modified} == true)
86  }
87  slot myNewInterestingPropertyNeedsFlushing = {property needsFlushing
88    <- eval ({this key} == {graControl interestingKey})
89    <- eval ({this owner} == systemID)
90    <- eval ({this modified} == true)
91  }
92
93  slot interestingPropertyIsNotUpdate = {property isUpdate
94    <- eval ({this modified} == true)
95    <- eval ({this isUpdate} == false)
96    <- eval ({this key} == {graControl interestingKey})
97  }
98  slot interestingPropertyIsUpdate = {property isUpdate
99    <- eval ({this modified} == true)
100   <- eval ({this isUpdate} == true)
101   <- eval ({this key} == {graControl interestingKey})
102 }
103
104 /*
105  * Add information relevant to GRA to new properties.
106  */
107 rule tagMyNewProperties 200
108 <- exists {property
109   <- eval ({this key} == {graControl interestingKey})
110   <- eval ({this owner} == systemID)
111 }
112 -> set myNewPropertySource = systemID
113 -> set myNewInterestingPropertyPositionX = systemPositionX
114 -> set myNewInterestingPropertyPositionY = systemPositionY
115 -> set myNewInterestingPropertyTimestamp = myNewInterestingPropertyTime
116 -> set myNewInterestingPropertyIsUpdate = false
117 -> set myNewInterestingPropertyNeedsFlushing = true
118
119 /*
120  * Remove properties that are outside their range of interest.
121  */
122 rule removeOutOfRangeProperties 160
123 <- exists {property <- eval ({this isUpdate} == true)}
124 -> retract {property
125   <- eval ((sqrt (
126       (({this positionX} - systemPositionX) ^ 2)
127       + (({this positionY} - systemPositionY) ^ 2)
128     )) > systemSensingRange)
129 }
```

```
130
131 /*
132  * Remove duplicates that occur when a node's information is send back to the
133  * node.
134  */
135 rule removeOwnDuplicateProperties 160
136 <- exists {property <- eval ({this isUpdate} == true)}
137 -> define tmp [duplicateID = {property id <- eval ({this isUpdate} == true)}]
138 -> retract {property
139   <- eval ({this id} == {tmp duplicateID})
140   <- eval ({this source} == systemID)
141   <- eval ({this owner} != {system owner})
142 }
143 -> retract {tmp}
144
145 /*
146  * Remove duplicates that occur when nodes retransmit the information they have
147  * from other nodes.
148  */
149 rule removeOtherDuplicateProperties 155
150 <- exists {property <- eval ({this isUpdate} == true)}
151 -> define tmp [duplicateID = {property id <- eval ({this isUpdate} == true)}]
152 -> set {tmp duplicateSource} = {property source <- eval ({this id} == {tmp
    duplicateID})}
153 -> set {tmp duplicateKey} = {property key <- eval ({this id} == {tmp
    duplicateID})}
154 -> set {tmp duplicateTimestamp} = {property timestamp <- eval ({this id} == {
    tmp duplicateID})}
155 -> retract {property
156   <- eval ({this id} == {tmp duplicateID})
157   <- exists {property
158     <- eval ({this source} == {tmp duplicateSource})
159     <- eval ({this key} == {tmp duplicateKey})
160     <- eval ({this timestamp} >= {tmp duplicateTimestamp})
161   }
162 }
163 -> retract {tmp}
164
165 /*
166  * Replace old properties with updated ones as other nodes send their updated
167  * properties over the network.
168  * NOTE: There is a crucial assumption in this code. At any time only one
169  * property fact can be a candidate for replacing another one. If there are two
170  * the second one is silently ignored and may well cause problems later on.
171  * This assumption is valid because having a new candidate implies that a new
```

```
172  * packet was received, and there can only be one single GRA fact per packet.
173  * After receiving a packet, the node MUST run the rule engine to process the
174  * new fact.
175  */
176 rule replaceUpdatedProperties 150
177 <- exists {property <- eval ({this isUpdate} == true)}
178 -> define tmp [updateID = {property id
179   <- eval ({this isUpdate} == true)
180   <- eval ({this source} != systemID)
181 }]
182 -> set {tmp updateSource} = {property source <- eval ({this id} == {tmp
        updateID})}
183 -> set {tmp updateKey} = {property key <- eval ({this id} == {tmp updateID})}
184 -> retract {property
185   <- eval ({this source} == {tmp updateSource})
186   <- eval ({this key} == {tmp updateKey})
187   <- eval ({this id} != {tmp updateID})
188 }
189 -> set {property isUpdate <- eval ({this id} == {tmp updateID})} = false
190 -> retract {tmp}
191
192 /*
193  * Cancel timeout below if the new value has changed to match the current
194  * value.
195  */
196 rule cancelTimeoutIfUnchanged 130
197 <- eval ({graControl timeout} > 0)
198 <- eval ({graControl newValue} == myRole)
199 -> set {graControl timeout} = -1
200
201 /*
202  * Update role once a random timeout has run out.
203  */
204 rule countdownRoleTimeout 120
205 <- eval ({graControl timeout} > 0)
206 -> set {graControl timeout} = ({graControl timeout} - 1)
207
208 rule setMyRoleAfterTimeout 120
209 <- eval ({graControl timeout} == 0)
210 -> set myRole = {graControl newValue}
211 -> set myRoleTimestamp = myRoleTime
212
213 /*
214  * Send (push) properties to other nodes.
215  */
```

```
216 rule propagatePropertiesInSensingRange 50
217 <- exists interestingPropertyIsNotUpdate
218 -> set interestingPropertyIsNotUpdate = true
219 -> send 0 systemTxRange interestingPropertyIsUpdate
220 -> set interestingPropertyIsUpdate = false
221
222 /*
223  * Remove the modified flag from facts that have been updated by the previous
224  * rule.
225  */
226 rule flushProperties 0
227 <- eval ({property needsFlushing} == true)
228 -> set {property needsFlushing <- eval ({this modified} == true)} = false
229 -> flush {property needsFlushing <- eval ({this modified} == true)}
230
231
232
233 /**
234  * Below is the Coverage specific part.
235  **/
236
237 // Only the role properties are worth propagating.
238 fact graControl [interestingKey = "role"]
239
240 /*
241  * Decide whether the current node should be in charge of covering a certain
242  * area...
243  */
244 rule coverageON 100
245 <- exists {property}
246 <- exists {property
247   <- eval ({this source} == systemID)
248   <- eval ({this key} == "temp-sensor")
249   <- eval ({this value} == true)
250 }
251 <- exists {property
252   <- eval ({this source} == systemID)
253   <- eval ({this key} == "battery")
254   <- eval ({this value} >= systemBatteryThreshold)
255 }
256 <- eval ((count {property
257   <- eval ({this source} != systemID)   // Don't count the node itself.
258   <- eval ({this key} == "role")
259   <- eval ({this value} == "on")
260 }) <= 1)
```

```
261 -> set {graControl newValue} = "on"
262 -> set {graControl timeout} = ((systemID * {system timeout-multiplier}) + {
        system timeout-offset})  // We use the system ID as timeout value for
        simplicity.
263
264 /*
265  * ... or not.
266  */
267 rule coverageOFF_1 100
268 <- exists {property}
269 <- exists {property
270   <- eval ({this source} == systemID)
271   <- eval ({this key} == "temp-sensor")
272   <- eval ({this value} != true)
273 }
274 -> set {graControl newValue} = "off"
275 -> set {graControl timeout} = ((systemID * {system timeout-multiplier}) + {
        system timeout-offset})  // We use the system ID as timeout value for
        simplicity.
276
277 rule coverageOFF_2 100
278 <- exists {property}
279 <- exists {property
280   <- eval ({this source} == systemID)
281   <- eval ({this key} == "battery")
282   <- eval ({this value} < systemBatteryThreshold)
283 }
284 -> set {graControl newValue} = "off"
285 -> set {graControl timeout} = ((systemID * {system timeout-multiplier}) + {
        system timeout-offset})  // We use the system ID as timeout value for
        simplicity.
286
287 rule coverageOFF_3 100
288 <- exists {property}
289 <- eval ((count {property
290   <- eval ({this source} != systemID)   // Don't count the node itself.
291   <- eval ({this key} == "role")
292   <- eval ({this value} == "on")
293 }) > 1)
294 -> set {graControl newValue} = "off"
295 -> set {graControl timeout} = ((systemID * {system timeout-multiplier}) + {
        system timeout-offset})  // We use the system ID as timeout value for
        simplicity.
```

Bibliography

[ADL+98] G. Asada, M. Dong, T.S. Lin, F. Newberg, G. Pottie, W.J. Kaiser, and H.O. Marcy. Wireless Integrated Network Sensors: Low Power Systems on a Chip. In *Proceedings of the 1998 European Solid State Circuits Conference*, 1998. http://www.janet.ucla.edu/WINS/download_publications/esscirc98.pdf.

[Bal98] Helmut Balzert. *Lehrbuch der Software-Technik*, volume 2. Spektrum Akademischer Verlag, Heidelberg, Berlin, 2nd edition, 1998.

[BFK+03] Carsten Buschmann, Stefan Fischer, Jochen Koberstein, Norbert Luttenberger, and Florian Reuter. SWARMS - Software Architecture for Radio-Based Mobile Self-Organizing Systems. Technical report, Institute of Operating Systems and Computer Networks, Technischen Universität Braunschweig, July 2003. http://www.ibr.cs.tu-bs.de/users/cbuschma/papers/Buschmann_etal_SWARMS.pdf.

[CGG+05] Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco. TinyLIME: Bridging Mobile and Sensor Networks through Middleware. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005)*, pages 61–72, Kauai Island, Hawaii, 2005. IEEE Computer Society Press. http://www.elet.polimi.it/upload/picco/papers/percom05.pdf.

[Dow04] Ian Downard. Simulating Sensor Networks in NS-2. NRL Formal Report 5522-04-10, Naval Research Laboratory, May 2004. http://pf.itd.nrl.navy.mil/nrlsensorsim/.

[DSV05] Adam Dunkels, Oliver Schmidt, and Thiemo Voigt. Using Protothreads for Sensor Node Programming. In *The REALWSN'05 Workshop on Real-World Wireless Sensor Networks*, Stockholm, June 2005. http://www.sics.se/~adam/dunkels05using.pdf.

[FR05] Christian Frank and Kay Römer. Algorithms for Generic Role Assignment in Wireless Sensor Networks. In *Proceedings of the 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys)*, San Diego, CA, USA, November 2005. http://www.vs.inf.ethz.ch/publ/papers/sensys05.roleassignment.pdf.

[GLvB+03] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of Programming Language Design and Implementation (PLDI) 2003*, June 2003. http://www.tinyos.net/papers/nesc.pdf.

[gre] Habitat Monitoring on Great Duck Island. Intel Research Laboratory at Berkeley. http://www.greatduckisland.net/.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987. http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf.

[hea] The Heathland Experiment Homepage. Telematics Working Group, Hamburg University of Technology. http://www.ti5.tu-harburg.de/projects/SensorNet/heathland.htm.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System Architecture Directions for Networked Sensors. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Cambridge, November 2000. http://www.tinyos.net/papers/tos.pdf.

[HV02] Annika Hinze and Agnès Voisard. A flexible parameter-dependent Algebra for Event Notification Services. Technical report, Freie Universität Berlin, 2002. ftp://ftp.inf.fu-berlin.de/pub/reports/tr-b-02-10.pdf.

[IGE00] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah 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, Massachussetts, August 2000. http://lecs.cs.ucla.edu/~estrin/papers/diffusion.ps.

[int05] Moore's Law – Made Real by Intel Innovation. Intel Corp., 2005. http://www.intel.com/technology/silicon/mooreslaw/.

[jes] JESS – The Rule Engine for the Java Platform. http://herzberg.ca.sandia.gov/jess/.

[JTC] ISO/IEC JTC1/SC22/WG17. ISO/IEC 13211-1 Programming Language Prolog. http://www.sju.edu/~jhodgson/wg17/.

[Kip92] James D. Kiper. Structural Testing of Rule-Based Expert Systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(2):168–187, 1992. http://portal.acm.org/citation.cfm?id=128894.128896.

[KK04] Vikas Kawadia and P.R. Kumar. Principles and Protocols for Power Control in Ad Hoc Networks. *IEEE Journal on Selected Areas in Communications: Special Issue on Ad Hoc Networks*, 2004. http://black.csl.uiuc.edu/~kawadia/papers/kawadia-kumar-jsac-camera-ready0.pdf.

[KK05] Vikas Kawadia and P.R. Kumar. A Cautionary Perspective on Cross Layer Design. *IEEE Wireless Communication Magazine*, 12(1):3–11, February 2005. http://black.csl.uiuc.edu/~prkumar/ps_files/cross-layer-design.pdf.

[KKT04] Ulaş C. Kozat, Iordanis Koutsopoulos, and Leandros Tassiulas. A Framework for Cross-layer Design of Energy-efficient Communication with QoS Provisioning in Multi-hop Wireless Networks. In *Proceedings of IEEE INFOCOM 2004*, Hong Kong, March 2004. http://www.inf.uth.gr/~jordan/INFOCOM-2004.pdf.

[KR05] Oliver Kasten and Kay Römer. Beyond Event Handlers: Programming Wireless Sensors with Attributed State Machines. In *4th International Conference on Information Processing in Sensor Networks (IPSN'05)*. UCLA, April 2005. http://www.vs.inf.ethz.ch/publ/papers/kasten-beyond-2005.pdf.

[LC02] Philip Levis and David Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, California, October 2002. http://www.cs.berkeley.edu/~pal/pubs/mate.pdf.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Pro-*

ceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003), 2003. http://www.cs.berkeley.edu/~pal/pubs/tossim-sensys03.pdf.

[mic] Datasheet of the MICA2DOT Wireless Microsensor Mote. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2DOT_Datasheet.pdf.

[Moo65] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, April 1965.

[MSK+05] C. Mallanda, A. Suri, V. Kunchakarra, S.S. Iyengar, R. Kannan, and Durresi A. Simulating Wireless Sensor Networks with OMNeT++. http://bit.csc.lsu.edu/sensor_web/final_papers/SensorSimulator-IEEE-Computers.pdf, January 2005.

[ns005] *The ns Manual*, June 2005. http://www.isi.edu/nsnam/ns/doc/ns_doc.pdf.

[Pie05] Thomas Pietsch. Entwurf und Implementierung einer grafischen Programmierumgebung für Sensorknoten in einem Funknetzwerk. Master's thesis, Department of Mathematics and Computer Science, Freie Universität Berlin, 2005.

[Pis] Kristofer S.J. Pister. Smart Dust – Autonomous Sensing and Communication in a Cubic Millimeter. http://robotics.eecs.berkeley.edu/~pister/SmartDust/.

[PSA92] A. Preece, R. Shinghal, and Batarekh A. Principles and Practice in Verifying Rule-Based Systems. *Knowledge Engineering Review*, 7:115–141, 1992. http://www.csd.abdn.ac.uk/~apreece/Pubs/KER92.html.

[RFMB04] Kay Römer, Christian Frank, Pedro José Marrón, and Christian Becker. Generic Role Assignment for Wireless Sensor Networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, pages 7–12, Leuven, Belgium, September 2004. http://www.vs.inf.ethz.ch/publ/papers/sigops.roleassignment.pdf.

[RKM02] Kay Römer, Oliver Kasten, and Friedemann Mattern. Middleware Challenges for Wireless Sensor Networks. *ACM Mobile Computing and Communication Review*, 6(4):59–61, October 2002. http://www.vs.inf.ethz.ch/publ/papers/wsn-middleware.pdf.

[sca] ScatterWeb Homepage. Computer Systems & Telematics Working Group, Freie Universität Berlin. http://www.inf.fu-berlin.de/inst/ag-tech/scatterweb_net/.

[SHM03] Ahmed Safwat, Hossam Hassenein, and Hussein Mouftah. Optimal Cross-Layer Designs for Energy-Efficient Wireless Ad hoc and Sensor Networks. In *The 22nd IEEE International Performance, Computing, and Communications Conference (IPCCC 2003)*, Phoenix, Arizona, USA, April 2003. http://www.cs.queensu.ca/home/safwat/samplepublications/Safwat_IPCCC_2003.pdf.

[SPMC04] Robert Szewczyk, Joseph Polastre, Alan Mainwaring, and David Culler. Lessons From a Sensor Network Expedition. In *Proceedings of the First European Workshop on Sensor Networks (EWSN '04)*, Berlin, Germany, January 2004. http://www.cs.berkeley.edu/~polastre/papers/ewsn04.pdf.

[Sto03] Jon "Hannibal" Stokes. Understanding Moore's Law. ars technica, February 2003. http://arstechnica.com/articles/paedia/cpu/moore.ars/.

[Tho99] Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley, second edition, 1999.

[TRV+05] V. Turau, Ch. Renner, M. Venzke, S. Waschik, Ch. Weyer, and M. Witt. The Heathland Experiment: Results And Experiences. In *Workshop on Real-World Wireless Sensor Networks REALWSN'05*, Stockholm, June 2005. http://www.sics.se/realwsn05/papers/turau05heathland.pdf.

[TS05] Kirsten Terfloth and Jochen Schiller. Driving Forces behind Middleware Concepts for Wireless Sensor Networks. In *The REALWSN'05 Workshop on Real-World Wireless Sensor Networks*, Stockholm, June 2005. http://page.mi.fu-berlin.de/~terfloth/mw_forces_realwsn05.pdf.

[TWS06] Kirsten Terfloth, Georg Wittenburg, and Jochen Schiller. FACTS - A Rule-Based Middleware Architecture for Wireless Sensor Networks. In *Proceedings of the First International Conference on COMmunication System softWAre and MiddlewaRE (COMSWARE)*, New Delhi, India, January 2006. http://page.mi.fu-berlin.de/~wittenbu/uni/facts/terfloth06facts.pdf.

[Win05] Rolf Winter. Personal conversation, September 2005.

[Wit] Georg Wittenburg. FACTS - A Rule-Based Middleware Architecture for Wireless Sensor Networks. http://page.mi.fu-berlin.de/~wittenbu/uni/facts/.

[WLLP01] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S.J. Pister. Smart Dust: Communicating with a Cubic-Millimeter Computer. *IEEE Computer Magazine*, pages 44–51, January 2001. http://doi.ieeecomputersociety.org/10.1109/2.895117.

[WLT] Georg Wittenburg, Matthias Lehmann, and Kirsten Terfloth. ScatterVM – A Virtual Machine for the ScatterWeb ESB. http://page.mi.fu-berlin.de/~wittenbu/uni/scattervm/.