DOCTORAL DISSERTATION

A Rule-Based Programming Model for Wireless Sensor Networks

Dissertation zur Erlangung des akademischen Grades einer Doktorin der Naturwissenschaften (Dr. rer. nat.) im Fachbereich Mathematik und Informatik der Freien Universität Berlin

vorgelegt von

Dipl.-Inform. Kirsten Terfloth

Datum der Disputation: 8. Juni 2009

Gutachter:
Prof. Dr.-Ing. Jochen H. Schiller, Freie Universität Berlin
Amy Murphy, Dr. Sc., Universita della Svizzera Italiana, Lugano

Abstract

Increased efforts towards device miniaturization have led to the emergence of a new class of ad-hoc networks, the so called wireless sensor networks. Individual devices or nodes are commonly battery-powered, small in size, equipped with a variety of sensors, frugal processing capabilities and a wireless transceiver. Spatially distributed within a deployment area, these nodes are able to autonomously form a network and cooperatively serve a specified task, for instance to acquire environmental data, to detect predefined events and/or to enable direct, physical interaction. Applications that rely on wireless sensor network technology are therefore typically concerned with the investigation of phenomena that either spread over a large area, that demand for autonomous scheduling over a great period of time, that require unobtrusive mechanisms for data collection or immediate reactivity to observed states. A wireless sensor network can hence be understood as an application enabler, a tool which can be utilized to build a specific application rather than having a purpose of its own.

Application development for these kinds of networks is however complex, error-prone and tedious: Resource scarcity, timing constraints and a typically asynchronous operational model inherent to embedded devices are directly exposed to a programmer while at the same time, the need to map application semantics to run on a distributed, unreliable network has to be objected. Instead of being able to implement the envisioned application in a problem-oriented manner, the developer is forced to take a system-oriented viewpoint. This circumstance is especially disadvantageous when considering application domain experts and not professional software developers to be prospective users.

This thesis proposes a holistic programming model called FACTS that combines two well-known mechanisms for abstracting from low-level challenges into a dedicated framework for wireless sensor network programming: Abstraction through provision of a better conceptual model via a higher-level language at design time, and abstraction due to deliberate support, especially at runtime.

First of all, FACTS increases the expressiveness of sensor networking concerns with the help of a domain-specific language. Event-centric, problem-oriented task specification is enabled relying on a rule-based programming paradigm, while at the same time accessible hardware-related functionality is limited to only relevant features. Reactivity is captured at the language level by means of utilizing a natural, declarative yet concise representation. Moreover, application knowledge can be denoted equally well with the help of rules, as has already been proven e.g. in the context of business rule specification, making rules a good choice for non-professional developers.

Furthermore, substantial support in terms of runtime support, development toolchain and encapsulation of typical sensor networking routines is provided within the FACTS middleware framework. The developer is empowered with a set of tools that accompany him throughout the development process and allow for simplified programming, debugging and testing. A core element here is the runtime environment that can be utilized on typical, small-scale wireless sensor nodes. It ensures the stable execution of rule-oriented programs by shielding a programmer from concerns such as manual stack management, correct event ordering and timing prerequisites of the underlying hardware. A number of protocols and applications ported to and developed for FACTS validate approach usability and shed a light on its advantages as well as on its bounds.

Acknowledgements

The work presented in this document was carried out with the support of a research fellowship within the Berlin-Brandenburg Graduate School on Distributed Information Systems at the Freie Universität Berlin, with some outstanding time spend at the University of Lugano, Switzerland.

I owe my sincere gratitude to a number of wonderful people that I had the pleasure to work, to discuss and to share "the good, the bad and the ugly" of being a PhD student with, and that generously lent me their time, eyes, ears and nerves during the time of working on this thesis. Here, I would like to take the opportunity to thank all of you for making this work possible.

First and foremost, I would like to thank my enthusiastic first advisor Prof. Jochen Schiller, who always emphasized the positive things, may it have concerned research questions, project work or leisure. The great time I have been allowed to spend at the Freie Universität Berlin would have never been as inspirational without his support, confidence and passion for new ideas. Having the opportunity to freely follow the research questions that really took my interest is a rare and precious gift I am especially thankful for being granted. The research group I was eligible to take part in has been more than just mere colleagues to me, and has made coming to the institute every day a joyful experience.

My special thanks go to Amy Murphy, D.Sc., who has not only been so kind as to review this thesis and discuss important parts, but also shared her vast knowledge, passion for abstraction and very valuable time in Lugano and Trento with me. Knowing how busy her schedule in general is, makes her accepting to be my second advisor a true honor. Furthermore, I would like to thank my second advisor from the Berlin-Brandenburg Graduate School on Distributed Information Systems, Prof. Oliver Günther, especially for the in-depth discussions within the semi-annual meetings of the GK, and the impact his remarks had at the beginning of this work. These stressful, yet very effective days have helped to put all the technical questions into a greater context and thus shaped the overall thesis tremendously. My thanks for the remarkable scientific feedback goes to all members and fellows of the Graduate School.

The research environment I found at the Freie Universität Berlin has been outstanding. Not only do I owe my peers from the Computer Systems and Telematics working group for the great, vivid surroundings they created, but also my diploma students for the hard work and effort that they put into the work presented within as well. Bootstrapping FACTS together with Georg Wittenburg has really been fun, and I owe him special thanks for the many things I have been able to learn from him as well as the friendship we share. Additionally, Freddy Lopèz Villafuerte deserves a big thank you for *always* being able to make me smile, and Anna Förster for the awesome experience in Lugano and for pushing the ScatterWeb nodes to their extremes with me.

I am very grateful to my friends that despite the fact of me being stressed out especially in the past year still keep me down to earth. Peter, Herr Oost, Vikki, it is great to just be around you! My very special thanks go to Katharina Hahn: Having a friend that not only is a wonderful person, but highly intelligent, with a great sense of humor *and* unconditionally sharing ones ups and downs in all respects can probably be found just once in life.

Olaf Dolfus deserves my deepest gratitude for being the incredible person he is. The love and support, patience and inspiration he has been given me over the last years cannot be thanked for in words.

Last, but not least, I want to express my gratitude to my parents, my beloved brothers, my family and family-soon-to-be. The degree of freedom, encouragement, support and love I have been privileged to enjoy throughout my entire life can doubtlessly not be taken for granted. I am very aware of how lucky I am, and want to thank you for always accompanying me on my way.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded devices are nowadays the insects of technology - there are simply more around than one would ever imagine! Already in the year 2000, out of all shipped processors 98% have been embedded CPUs, preferably microcontrollers with an 8- or 16-bit datapath and extremely limited RAM sizes [135]. Commonly found for instance in automation, vehicles, consumer electronics or medical equipment, these processors are designed to run specific tasks instead of serving as a general-pupose computing device and dissolve in their host, not being visible to a user. As size and production cost matter since these are the critical parameters for market success and obtainable profit, on-chip resources are usually constraint to the absolute minimum.

With the emergence of so called smart objects, thus objects of everyday life empowered with computational and possibly communicational capabilities, the border between pure embedded devices and traditional, computational rich systems clearly blurs. These devices are envisioned to be ubiquitous and inexpensive, yet expected to be reconfigurable and easy to adapt. Since the mere number of electronic objects to handle has dramatically increased over the past years, usage and maintenance convenience have become ever more important for purchase decisions. Appropriate means to control smart items are mandatory for their fast adoption. A typical product that resembles these demands is the ePaper technology embedded in a corresponding reading device [63]: this simple electronic item mimics an everyday item (regular paper), seeks to provide its key advantages over already available displaying technologies (greater angle of vision, reflective instead of transmissive display and high contrast) and enhances it with the capability to hold libraries of books on a chip. The deliberate design decision for less functionality (no support for multi-media) in favor of lowest possible power consumption points out another observable trend for increased emphasis on energy-efficiency: A new sensitivity towards environmental concerns and a fatigue of a constant need to recharge all kinds of objects has triggered a movement for *green IT* - resource-efficient, environmentally compatible technology that nevertheless provides up-to-date functionality.

A class of devices that is on the verge of the embedded domain are *wireless sensor networks (WSNs)*. Battery-powered and small in size, individual devices, so called *wireless sensor nodes*, are spatially distributed to autonomously form an ad-hoc network. Each node is commonly equipped with a variety of sensors to sample data of physical phenomena, a wireless transceiver to interoperate with other sensor nodes in the network and frugal processing abilities. Rather than being objects of everyday life, they serve as a tool for distributed data acquisition, event detection or direct, physical interaction. A wide range of different applications can hence instantly benefit from depending on this technology. Usually, these applications are concerned with the investigation of phenomena that either spread over a large area, that demand for autonomous scheduling over a great period of time, where the intervention of taking data samples has to be non-invasive, or that depend on immediate reactions to a specific state which is possibly acquired from distributed data samples.

In the remainder of this chapter, we present the motivation for introducing a holistic, rule-based programming framework to enable improved sensor node tasking in Section 1.1. Afterwards, we will point out the main contributions of this thesis in Section 1.2, before turning to the actual structure of this work in Section 1.3.

1.1 Problem Statement

Within the above section, the statement has been made that smart objects, and in particular wireless sensor networks, are at the verge of the embedded domain. From a system-oriented point of view, this is not evident as they depend completely on embedded hardware. However, unlike embedded devices that are in general build and programmed to satisfy a single purpose, wireless sensor network capabilities allow for their application in a diversity of problem domains, making them a valuable tool in miscellaneous settings. As such, wireless sensor networks have e.g. been successfully applied by biologists to monitor chemical processes within redwood trees [140] or the habits of vulnerable animals [100, 106], by geologists to investigate the development of glacial regions [103], for object [11] and asset tracking of freight containers [119], but also in home automation [70], to enable event-detection in general [122] or for emergency response in medical applications [59].

Clearly, the application logic, thus the actual task that is determined to be running on the sensor network, will typically be defined by the corresponding domain experts. Only they have the knowledge to specify what kind of data has to be taken and at what rate, how to process the acquired samples, what states have to be monitored and how to react appropriately in respect to different situations. Since requirements may change over the time of network deployment, especially when utilized in a setting that is motivated by a research question, network reconfiguration and/or retasking has to be made available. These requirements totally

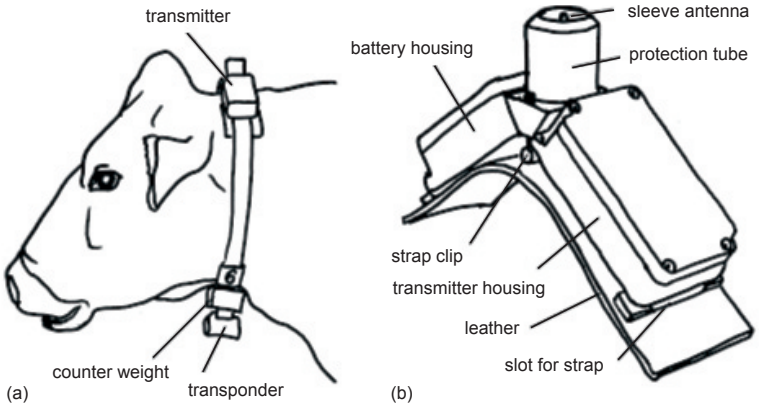


Figure 1.1: Collar for dairy cows (a) Transponder at the neck of the cow and (b) main external parts of the transponder [taken from [68]]

differ from general embedded device utilization patterns: Here, devices are programmed by professional developers, implemented firmware is flashed once onto a device and will afterwards never be touched again. Given approximately the same hardware resources but a different background and expertise in embedded programming, it becomes evident that domain experts will face a number of concerns when trying to map envisioned network behavior into software.

In the following, a somewhat extraordinary but valid, real-world example for a possible wireless sensor network application will be illustrated to point out, what in detail the core challenges are that developers face when trying to implement wireless sensor network applications.

1.1.1 Cow tracking to assist studies on social interaction patterns for increased dairy productions

In agricultural dairy farming, the condition and well-being of a herd has a great impact on the health of individual animals, and in turn on their actual productivity. Lately, veterinary research has especially focussed on studying social behavior and interaction patterns of dairy cows [57] in order to relate findings to the occurrence of diseases that inhibit optimal stock breeding conditions. To assist these efforts, an exact mechanism to position individual cows within a stable, but also outside in the meadows has to exist, as well as a backbone system that is able to process acquired data. One possibility to set up a local indoor position measurement system has been presented in [68], where radar transponders have been integrated into a collar worn by each cow, see Figure 1.1. Since the antenna for the transponder is mounted high on the animal, signal shading and masking by the animals' body can be prevented. A corresponding reception system built from

several basestations that listen to emitted data have been installed at the stable ceiling to enable minimal signal disturbance. Positioning is then implemented relying on a time of arrival approach, relating emitted radar signals received at various base stations to one another to calculate a cows relative position [81].

The prime goal of the system setup presented above targets pure data acquisition and offline data evaluation as no means for direct interaction are available. The system itself is passive. However, in case transponder and basestation nodes become smart, thus are exchanged by wireless sensor nodes that feature on-node data processing capabilities, the complete system is able to become (re)active [129]. A whole new spectrum of possible applications can then be enabled, ranging from the implementation of simple alerting strategies in case animals do not get enough water over the course of a day, refuse or are unable to move according to usual animal movement patterns or are simply sensed to be sick due to subsequent measurements of increased body temperature, up to sophisticated direct interaction. Concentrated pellets enhanced by supplementary vitamins or minerals may be fed automatically to cows that have an increased demand due to their individual health condition or social grouping of specific cows may be enforced or prevented by means of controlling gates and passages to name but a few scenarios that are conceivable.

To implement necessary data acquisition patterns, control mechanisms and reactions according to predefined animal conditions, an agricultural farmer or a veterinary has to denote the corresponding rules that capture each target situation in combination with the envisioned system behavior and map this application knowledge to work on the embedded sensor network platform.

1.1.2 Challenges for wireless sensor network programming

While the first step, the clear specification of system behavior, can already be time-consuming but manageable even by non-expert programmers, the second step of transferring the application knowledge to a wireless sensor network application is tedious and challenging. As has been stated before, the target platform consists of embedded devices, exporting a low-level of abstraction from system properties such as resource scarcity, both in terms of available energy and memory, limited processor performance and available interfaces for debugging and control, the typically asynchronous operational mode as well as the need to satisfy timing constraints directly to the programmer. For reasons of increased flexibility in regard to deployment strategies, sensor nodes are equipped with wireless networking capabilities. However, wireless links are unreliable, so that additional attention has to be payed to overcome inconsistent states among sensor nodes due to lost packages. Finally, wireless sensor network applications often take a network-level view upon problem definition, thus understand the network itself as a tool to implement the solution to the specific assignment in question. Coordination of nodes is therefore mandatory to achieve the envisioned behavior. Years of research have already pointed out that the development of applications depending on distributed

networks is cumbersome and calls for a thorough understanding of underlying network implications.

In summary, the following properties of wireless sensor networks render software development to be a challenging task:

- *Embedded hardware*: Since wireless sensor nodes build upon simple microcontroller architectures to optimize for cost and size, system properties such as resource scarcity and the corresponding demand for energy-efficient tasking, interrupt-driven execution and timing concerns are visible throughout the software stack and have to be handeled appropriately. The interface for software development therefore enforces a system-oriented point of view, offering only a low level of abstraction to a software developer.

- *Asynchronism*: The preferred execution model for wireless sensor network processing tasks depends on an event-driven control loop. This choice is motivated by the quest for maximizing energy efficiency, allowing a node to return to a low-power idle mode in case no actions are required. While this strategy allows to substantially increase node lifetime, it shifts the burden of coping with asynchronism to the programmer. Push semantics which are not captured well with predominantly used imperative programming models have to be integrated into program flow.

- *Wireless networking*: The flexibility of wireless networking comes at the cost of unreliable network links. In addition, wireless communication is also costly in terms of energy spent on both sending and receiving nodes. Therefore, from a programming perspective, considerable effort has to be put into balancing the urge for reliable communication and energy-efficient software implementation.

- *Distribution*: Due to a foremost network-level conception of wireless sensor network applications, coordination and synchronization of groups of nodes are often mandatory for successful execution. This however is, especially in respect to the afore mentioned costly and unreliable means for communication, a challenging demand programmers have to face.

If we now relate the denoted challenges of the WSN domain to the expectations a potential user has regarding future applications, the actual problem that requests a sensitive, yet powerful intervention as visualized in Figure 1.2 becomes evident.

On the one hand, one can draw from the example sketched in Section 1.1.1 that a domain expert comprehends her envisioned application in terms of the *problem* that needs to be solved. Interaction and reaction patterns are expressed with network behavior in mind and the focus is clearly to capture correct application semantics. The taken viewpoint is thus problem-oriented.

On the other hand, the target platform that enables to build powerful, distributed applications in a low-cost manner is extremely challenging in respect to

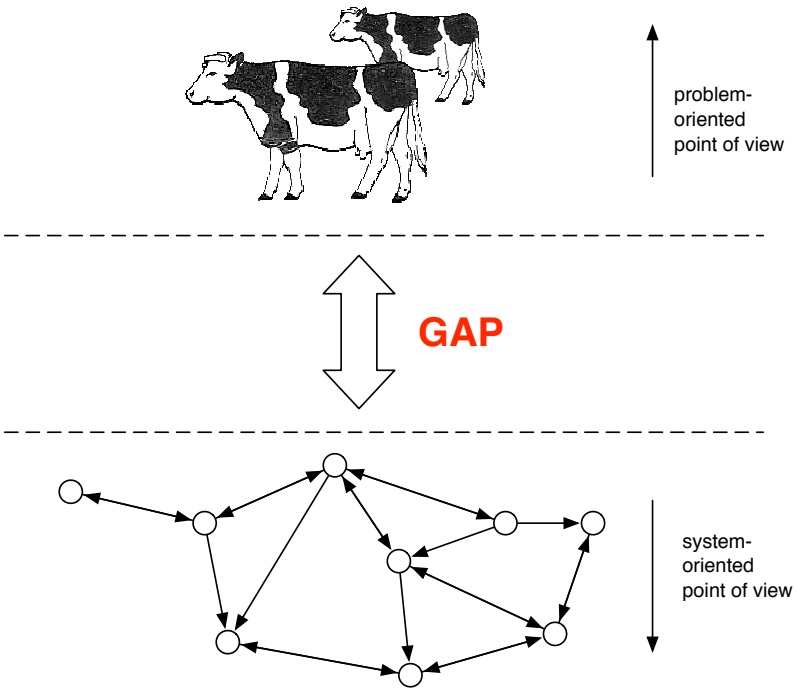


Figure 1.2: Imbalance of viewpoints for application development in wireless sensor networks, see also [120]

the demand for expertise knowledge in software development. A low level of abstraction due to relying on embedded, distributed devices that interoperate with the help of wireless communication imposes a system-oriented point of view on corresponding programmers.

The central concern of this dissertation is to close the gap situated in between the problem- and the system-oriented world.

1.2 Contributions

This thesis proposes a holistic programming model called FACTS that combines two well-known mechanisms for abstracting from low-level challenges into a dedicated framework for wireless sensor network programming:

First of all, FACTS increases the expressiveness of sensor networking concerns with the help of a dedicated, domain-specific language. Event-centric, problem-oriented task specification is enabled by relying on a rule-based programming paradigm, while at the same time accessible hardware-related functionality is limited to only *relevant* features. Reactivity is thus captured at the language level by means of utilizing a natural, declarative yet concise representation. Moreover,

application knowledge can be denoted equally well with the help of rules, as has already been proven e.g. in the context of business rule specification.

Furthermore, substantial support in terms of runtime support, development toolchain and encapsulation of typical sensor networking routines is provided within the FACTS middleware framework. The developer is empowered with a set of tools that accompany him throughout the development process and allow for simplified programming, debugging and testing. A core element here is the runtime environment that can be utilized on typical, small-scale wireless sensor nodes. It ensures stable execution of rule-oriented programs by shielding a programmer from concerns such as manual stack management, correct event ordering and timing prerequisites of the underlying hardware. A number of protocols and applications have been ported to and developed for FACTS to validate approach usability and shed a light on its advantages as well as on its bounds.

This combination of language, development and runtime support allows for a shift of the developers' focus from platform-specific concerns back to the application semantics.

Overall, the contributions of this thesis can be summarized as follows:

- *Qualitative analysis of the current state of the art*: Numerous abstractions to overcome the intrinsic challenges of WSN programming concerns have been proposed so far. A set of conceptual and functional criteria has been compiled to analyze the most prominent and distinctive approaches and provide a means for their evaluation.

- *Specification of a suitable and concise language for WSN tasking*: The approach proposed in this thesis evolves around the rule-based programming language RDL, which has been especially crafted to suit WSN challenges. Motivating the reasons for conceptual design decisions and implemented functionality, language syntax is denoted and semantics presented in a formal manner. To evaluate language utility, thus to provide a first impression on its pragmatics, examples for typical usage scenarios as well as paradigm shortcomings are critically discussed.

- *Provision of a versatile programming framework*: The FACTS middleware framework comprises a set of tools to aid in the development process, amongst them different backends to which RDL rules can be compiled to. Powerful support during the test- and the deployment phase enable the implementation of well-crafted, stable and scalable sensor network protocols and applications that run equally well in simulation and real-world environments. Furthermore, the design space for system-related access to hardware has been explored, and an elegant means for low-level configuration and control has been integrated into the framework.

- *Evaluation of optimization strategies towards better memory consumption and increased runtime performance*: Adding an additional layer of software in between system and application and depending on interpretation

instead of native execution automatically results in performance loss. To lower this impact, optimization strategies targeting better memory utilization and faster rule interpretation have been explored and quantified.

- *Extensive testing of approach validity by implementation of a variety of protocols and applications*: A great number of typical sensor networking tasks has been implemented on the FACTS middleware framework, amongst them two prominent routing protocols and a real-world application. These have been utilized on the one hand to verify that qualitative goals have been successfully met, on the other to quantify their performance.

1.3 Thesis Overview

This thesis comprises eight chapters that present, analyze and evaluate the proposed middleware framework FACTS, and put the corresponding developments into the context of current research directions. Following this introductory chapter, key technologies, application areas and widely adopted software stacks for wireless sensor networks, as well as basic approaches to capture event-centricity and to develop reactive software are presented in Chapter 2. General terms, definitions, concepts and keywords that are utilized throughout the thesis will be clarified here, providing a reader with the necessary background.

A more specific viewpoint concerning the problem of abstraction provision for wireless sensor networks will be taken in Chapter 3. Therefore, a representative excerpt of current approaches and models is categorized by their specific perspective on valuable abstraction methodology. In order to enable a thorough qualitative analysis, basic idea and implementation are presented for each approach, before functional demands derived for wireless sensor network abstractions in general are used to judge the individual focus of the corresponding abstraction.

Chapter 4 finally turns to the main part of this thesis. Since the FACTS middleware framework comprises several components to be discussed in subsequent chapters, the intention of this chapter is to provide a brief overview of functional parts, thus allow for a first, high-level impression. Moreover, the set of metrics developed and applied in Chapter 3 to similar abstraction approaches are used to point out FACTS qualities and its prime abstraction goals.

FACTS exports a rule-based programming paradigm to serve as a means for wireless sensor network protocol specification. A comprehensive introduction of the therefore developed ruleset definition language (RDL) is the concern Chapter 5. After a presentation of the rather concise RDL syntax, a formal specification by means of denotational semantics provision follows. The picture is completed by a discussion of language pragmatics: Programming patterns, derived from various rule-based implementations, that on the one hand tackle intrinsic shortcomings of event-driven application development and on the other feature common WSN data processing approaches, enable an abstract conception of language features.

Design and implementation of the functional parts of the FACTS middleware platform are discussed in Chapter 6. These comprise the ruleset compiler, which maps the problem-oriented specification of node-local behavior to an executable bytecode and a corresponding runtime environment in charge of interpretation. Target platforms for the compiler subsume a functional simulation environment for improved, fine-grained protocol debugging, the MSB430 ScatterWeb platform for real-world deployments and the `ns-2` network simulator simulating a network of ScatterWeb sensor nodes. Furthermore, an approach for a lightweight integration of configuration and control issues for underlying hardware capabilities is introduced, exploiting available middleware abstractions. Where applicable, optimization schemes and their impact are presented and quantified.

Chapter 7 is devoted to demonstrate in detail how the FACTS programming framework works in practice. Therefore, system-level and application rulesets have been implemented to explore the design space of RDL programming capabilities. Middleware functionality available for re-use comprises e.g. two established routing protocols especially designed for the wireless sensor networking domain. Both have been implemented in a native and in a rule-based manner and compared concerning their performance in regard to usual protocol metrics as well as to quantify the impact of interpretation. A prototype application that has been successfully simulated to run on the FACTS framework is concerned with a monitoring task for a fence and implements event detection mechanisms respectively.

Finally, Chapter 8 wraps up this thesis and points out distinct findings and the main contributions. A small outlook on future concerns and unsolved, but relevant problems that need to be addressed to put the vision of ubiquitous, technological progress accessible to a vast audience to practice conclude this thesis.

Chapter 2

Background

Holistic wireless sensor network tasking concerned with the provision of both a practical conceptual model at design time and a powerful execution environment at runtime requires a broad foundation within software development. Therefore, this work has been inspired by a variety of existing methodologies and concepts stemming from and evaluated within different areas of computer science, transformed them into suitable tools for the target domain and fused them into a comprehensive framework.

This chapter provides general information on research areas that substantially influenced this work, namely the state of the art in wireless sensor networking and concepts and practice for event processing. Hence, it introduces definitions and terms drawn from these domains, clarifies basic technological aspects that this thesis is based on and presents core directions that developments may have targetted. Rather than dwelling in details, the interested reader will be redirected to relevant work with pointers to the corresponding research papers where appropriate.

The organization of this chapter is as follows: Section 2.1 features a brief introduction to wireless sensor networks. Besides providing a general description of technological aspects, software stack and application areas, this section will highlight the challenges that differentiate these networks from other ad-hoc networking approaches in more detail. Since the approach this thesis suggests to adopt is a representative of event-driven architectures, Section 2.2 will clarify relevant terminology this domain utilizes and present common classes of event-centric architectures. Within this scope, Section 2.2.4 is especially concerned with rule-based approaches. With a brief introduction to rule-based programming and a differentiation of available dialects, their implications on application areas and implementation, this section renders a picture of substantial efforts undertaken in this area. Section 2.3 finally discusses presented findings and wraps up this chapter.

2.1 Wireless Sensor Networks

Technological progression and its massive application in both industrial and personal life revolutionized everyday processes in industrial countries over the last two decades. Mass production of personal computers, handhelds and cell phones in combination with a steady increase in availability of wireless connectivity, better service provision and richer device functionality have lead to their wide acceptance and almost seamless integration into many aspects of human being life. Corresponding wireless networking technologies and standards such as Wireless LAN, UMTS or GSM are typically operated in an *infrastructure-based* manner and provide a central access point that coordinates network setup, medium access among participants and forwards data to its intended recipients.

Decentralized wireless networks, or *ad-hoc networks*, are in contrast networks that depend on mechanisms to self-organize coordinated network behavior. The assignment of which node is in charge to forward packets for its neighboring nodes is not fixed but has to be adapted dynamically with respect to network connectivity. As nodes may be mobile and wireless link quality may vary over time, network topology is likewise subject to change. Ad-hoc networks are especially valuable in situations or regions where infrastructure-based networks are either too costly to deploy and maintain (e.g. for large scale and/or temporary environmental monitoring), when their setup time is too long for the requested operational setting (e.g. after a natural disaster) or spontaneous meshes are sufficient (e.g. to build a personal area network for the exchange of electronic vCards).

The tremendous efforts put into minituarisation of integrated circuitry has led to shrinking sizes of wireless technology within the last few years and gave rise to the vision of so called Smart Dust [145]. These tiny devices envisioned to be the size of a dust particles, are able to communicate with each other, to sense their environment and to locally perform basic processing on acquired data, can be distributed in large quantities. In general, these *wireless sensor networks (WSNs)* come in different sizes, ranging from developments of nodes manufactured as a system-on-a-chip (SoC), e.g. Smart Dust, over practical-sized platforms that depend on components available commercially of-the-shelf (COTS), e.g. the Mote family [78], up to huge, specialized single-pupose nodes, e.g. the SOSUS platform [151]. All of these however have in common that they are deployed close to physical phenomena which are to be observed by means of sampling available sensors and that they communicate with each other autonomously to self-organize the desired network behavior. As a consequence, the number of nodes that form a WSN depends on the range a corresponding sensor can cover, the area over which the observed aspect spreads and demanded accuracy, and may range from a couple of nodes up to several hundert.

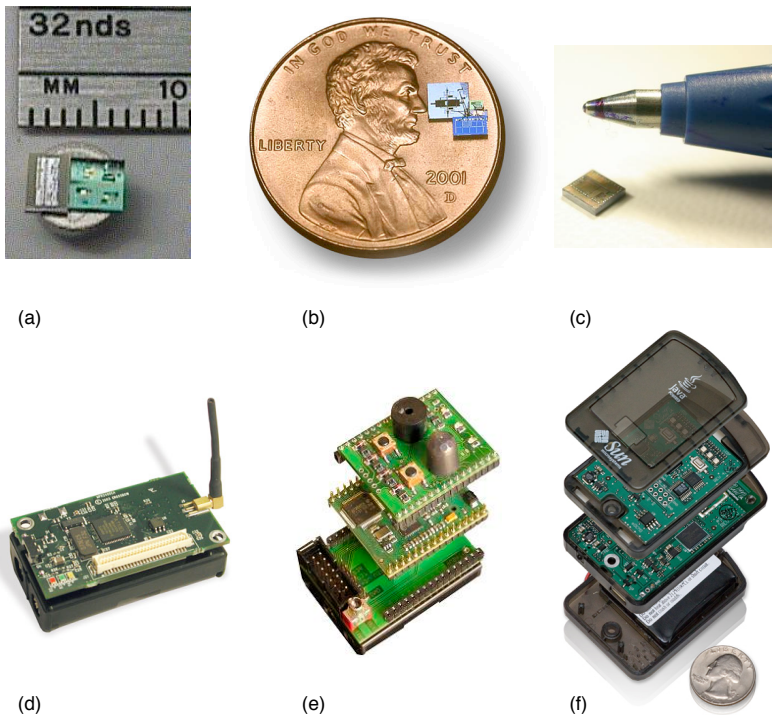


Figure 2.1: Wireless sensor network platforms (a) Smart Dust [114] (b) CCR node [114] (c) Spec node [77] (d) MicaZ platform [39] (e) MSB430 modular sensor board [14] (f) SunSpot platform [2]

2.1.1 Platforms

The design and assembly of wireless sensor node platforms has received a lot of attention. An exemplary overview of a few, popular sensor nodes is presented in Figure 2.1 where the upper row depicts prototypes that conform to a SoC design and the lower presents those that have been put together from commercially available parts. For instance, 2.1 (a) shows a Smart Dust node developed in 1999, which gave rise to the vision of invisible, low-cost data acquisition. This platform, mounted upon a button cell and featuring a tiny, digital controller, as well as the CCR node depicted in 2.1 (b), depend on optical data transmission: Light emitted by a laser from a basestation allows for communication to the nodes, whereas the motes communicate in backward direction by modulating their reflectivity using a so called MEMS corner cube [86].

Optical communication requires a line of sight in between communication partners, which is, in combination with the necessary communication initiation by a basestation, a fairly strong restriction. Hence, communication based on radio frequency transmission is nowadays preferred. Platforms depicted in 2.1 (c) - (f)

therefore all feature a radio transceiver, allowing for omnidirectional communication thus increased flexibility for deployment strategies. On the downside however, these systems have a higher energy consumption due to geometrical signal attenuation.

While the goal for maximal reduction of physical sensor node size has been a driving force at first, for a lot of experiments, mid-size platforms prove to be sufficient. Moreover, node size is also predetermined by the energy requirements for specific deployments, which in most cases are a lot higher than what a button cell can provide. Naturally, this dependency triggered a lot of interesting research in the area of energy harvesting and its applicability within the WSN domain, which is discussed in detail in [125] and left out here for brevity.

The majority of platforms are larger size COTS platforms as presented in the second row of Figure 2.1, such as e.g. the Berkeley Mica mote family [79], the ScatterWeb MSB430 platform [14, 96], both representatives of research initiatives, or the Sun Spot platform [2] as a commercial product. Other system architectures not displayed here include BTnodes [21], Smart-Its [17], and for instance Particles [42]. These devices are typically equipped with a reprogrammable 8- to 32-bit microcontroller in charge of controlling the sensors, peripherals, eventually the RF radio and of executing the current application. Memory sizes are typically around 5KB for volatile SRAM and 55KB-128KB for non-volatile Flash memory. Additionally, secondary storage may be available, e.g. provided as EEPROM memory or via an external SD-card. Chosen frequency band and thus transceiver capabilities vary amongst the different architectures; especially in a research setting, RF radio transceivers which operate in the 868MHz and the 443MHz ISM band are preferably chosen, but platforms utilizing the license-free 2,4GHz frequency band can equally be found, see also Table 2.1 for a first impression. Available sensors are often supplied on an additional board, which can be adapted due to application concerns and are therefore not listed here. A detailed overview of existing popular WSN platforms and their individual setup can be found in [15] and an evaluation of a small subset including metrics to judge them in [20].

2.1.2 Application Scenarios

Wireless sensor networks are a versatile tool for in-situ data acquisition and immediate, localized reactions. Since the deployment itself can be literally untethered from any infrastructural requirements, the set up of a working system is very flexible, and therefore especially beneficial for short term deployments, for deployments that lack existing infrastructure to build upon (e.g. in wildlife) or demand for constant adaptiveness due to e.g. node mobility. Applications and studies carried out with WSN technology so far either focus on constant, often long-term data collection or they are set up for event detection and immediate reporting. Note that this classification is not exclusive (meaning that often times, elements of both design goals are combined) but rather yields to illustrate the extreme ends of the design space for WSN applications.

Table 2.1: Wireless Sensor Networking Platforms

	Mica2Dot	MicaZ	MSB430	Sun Spot
Microcontroller	ATmega128L	ATmega128L	TI MSP430F1612	ARM7
Architecture	8-bit	8-bit	16-bit	32-bit
SRAM	4KB	4KB	5KB	256KB
Flash	128KB	128KB	55KB	2MB
User Interface	3 LED	3 LEDs	1 LED	2 LEDs
Radio	CC1000	MPR2400	CC1020	CC2420
RF band	315-916MHz	2,4Ghz	402-915 MHz	2,4GHz

The prime goal of monitoring application deployments carried out in the past has been to observe a specific phenomena to control or understand (natural) processes. Representative approaches have been concerned with studying the habits of animals such as zebras, birds or cattle [98, 100, 68], with observing environmental states of e.g. glaciers, mountains or volcanos [103, 134, 149] or with the quest for structural monitoring of e.g. bridges, buildings or dikes [89, 29, 127]. Often, required data rates are rather low, whereas deployment time can span over a long period of time.

Military, medical or logistics applications [11, 118, 59] commonly put their emphasis on the detection of *specific* data items or network states to either invoke a localized reaction or to emit an early warning about a critical situation. In case a predefined event is likely to occur, sampling frequencies of sensors involved in phenomena detection are increased so that accurate, fine-grained data is available about the event in question if necessary.

In the following, an exemplary application from both ends of the spectrum is sketched in more detail to convey a feeling for design goals, setup prerequisites and system sizes.

Environmental research: Redwood Tree Monitoring

Environmental processes and developments are complex research subjects. For a thorough understanding of observed phenomena, an environmental researcher has to examine a number of different parameters which may influence the subject of interest, put these into relation to one another and finally develop a hypothesis to be verified. However, data collection is not always easy as traditional hardware may be too intrusive, too expensive or the area to cover too vast or inaccessible.

One of many deployments motivated by a biological research question carried out with wireless sensor networks has been a set up within a redwood tree to monitor its micro-climate [140]. Variations in temperature, humidity and light are known to appear over time and spatial distribution in a tree, but exact, concur-

rent measurements have been missing. A wireless sensor network consisting of 33 Mica2Dot nodes has therefore been deployed within a coastal redwood canopy at different heights and positions with respect to the stem of the tree. Over a period of 44 days, each sensor node was requested to measure temperature, relative humidity and light emission every 5 minutes, log the data locally and forward it in a multi-hop manner to a dedicated gateway of the network. Overall, the system was able to acquire multi-dimensional data as envisioned, however data yield, thus data delivery performance, has been disappointingly small as nodes died and logs filled early due to mismanagement. Nevertheless, the project revealed the feasibility of sensor networking technology for monitoring tasks as well as common pitfalls that need to be taken into account ahead of deployment time.

Vehicle tracking: The VigilNet Project

The design of a system for surveillance with applications such as vehicle tracking and event detection with the help of wireless sensor networking technology has been the goal of the VigilNet project [73]. Relying on 70 sensor nodexs, the general objective has been the provision of an early-warning mechanism which is able to detect and track a vehicle within the deployment area at reasonable precision and confidence. To furthermore guarantee both, lifetime of the complete network and latency of event reporting, to stay within acceptable bounds, a sophisticated software architecture to coordinate network-wide duty cycling and time synchronization has been developed. The network is split into distinct regions, each establishing a backbone to a predefined relay node and stable sensing coverage, before uninvolved nodes may put themselves to sleep and sensing by only a designated section starts. In case an event, thus a vehicle, is detected, sensor nodes in the corresponding target area are awakened and fine-grained data collection is triggered. Simultaneously, reports on the tracked vehicle are constantly sent to the a central entity in charge to allow for an adequate reaction.

2.1.3 Operating Systems and Protocol Stack

Not only a large number of different sensor networking platforms are by now available, but also a variety of operating systems that a programmer can rely on. Networked, resource-constrained devices are, for reasons of energy-efficiency and small memory footprint, often operated in an event-driven manner which is typically mirrored by the OS. Besides the provision of a convenient and safe abstraction from hardware resources, including timers, memory, basic communication and sensor primitives, and eventually additional service support, an OS therefore also exports its design philosophy to a programmer. Nowadays, a number of operating systems especially designed for wireless sensor networks exist such as for instance TinyOS [4], Manits [22], Contiki [46] and SOS [72]. To showcase the design space of existing developments, the following will feature a concise overview of TinyOS, Contiki and the ScatterWeb hardware abstraction layer that FACTS utilizes.

TinyOS

TinyOS, with its wide-spread usage, active community, open-source availability and magnitude of projects relying on its abstractions, is the *de facto* standard operating system for wireless sensor networks. It promotes a component-based model for organizing software parts by enforcing modularity at design-time instead of relying on the usual structure of a layered architecture. Components can e.g. wrap hardware functionality or application-level code and make their implementation then available via interfaces which may in turn be *wired* together for application composition.

Concurrency is handled in TinyOS with two distinct mechanisms, namely *tasks* and *events*. A task may be *posted* by a component and will run to completion before the thread of control is returned to its originator. The TinyOS task scheduler implements a FIFO queue to process incoming tasks according to their ordering. To ensure responsiveness of the system, tasks should therefore contain a non-blocking, short-running set of instructions. On the other hand, events, which typically represent hardware interrupts, may preempt tasks and events, and also run to completion. In case a long-running operation has to be triggered by a component, a split-phase operational model is suggested. The invocation of a so called *command*, declared in the public interface of a component, will start the corresponding execution and an event is raised upon completion to signal this to the calling component. Consequently, it has to implement an event handler to be executed thereafter. The flow of control is thus handled via commands in a top-down, and notification via events in a bottom-up direction. Programmers utilizing TinyOS are directly exposed to the impact that long-running executions have on the system and have to address this within their sourcecode - traditional, sequential program flow is hence not simulated by TinyOS.

With the provision of *active messages*, TinyOS mimics the port concept utilized by the TCP/IP stack: Each message is tagged with an identifier which is mapped to a specific handler on the reception side of the message. Simple unicast and broadcast functionality for one-hop communication is wrapped in the *GenericComm* module, thus can be wired to an application. Furthermore, the hardware abstraction consists of a set of core components that encapsulate e.g. microcontroller usage or clocks and timers, exporting a corresponding interface to make them accessible.

TinyOS components have to be implemented in the nesC programming language, an extension of C that enables the separation of concerns into individual components [61]. However, many features that C offers to a programmer, such as dynamic memory allocation, the specification of pointers, etc., are suppressed by nesC in order to improve execution safety. Since other common problems such as deadlocks and race conditions can however occur during execution due to the supported concurrency model, this static language design can be used for compile-time analysis. Besides the provision of core OS functionality, an extensive toolchain, including a simulator and a huge code base, is available for TinyOS.

Contiki

Contiki is the runner-up in the world of operating systems for wireless sensor networks. The driving force for the development of this OS has been to make it an open-source, portable, multi-tasking OS that can be utilized on a number of systems even with a different hardware background. Written in the C programming language, Contiki revolves around an event-driven kernel with three concurrency models supported: *events, threads* and the so called *protothreads*. In general, asynchronous processing requests, e.g. interrupts, are encapsulated in non-preemptable events, dispatched to running processes by an event scheduler, which then run to completion. Since stack memory can be regained after an event handler returns, the kernel may use a single shared stack for process execution. In addition to asynchronous event processing, the scheduler allows for interleaving it with periodical polling. Upon invocation, all processes (usually processes operating close to the hardware) that implement a poll handler are called, which allows them to e.g. check the status of hardware devices.

Support for the specification of multi-threading programs is available with Contiki as a library that may optionally be linked. Since the demand for a separate stack per thread and the need for locking mechanisms to access shared resources are inherent to the multi-tasking operational model, additional overhead incurs with its utilization. Stack has to be allocated prior to thread execution and cannot be shared thereafter until the thread returns, making this concurrency model quite costly in terms of memory consumption, a circumstance not always feasible for every application or platform.

A unique mixture of events and threads offered by Contiki are the protothreads. As opposed to usual threads, a protothread is stackless, thus does not have a history of function calls and can not, as a consequence, invoke a blocking function. It implements a conditional, blocking wait statement to be checked whenever the function that specifies it is invoked. In practice, this allows a function to run up to a certain point in a program, return the control flow to the scheduler and release its stack, then wait for an event to occur and return to exactly the previously stored location of execution within the function when being invoked the next time. In case additional stateful information for context conservation is needed, this has however to be stored in global variables.

Naturally, a number of core OS services are available in Contiki, including a set of modules for general hardware abstraction, implementations for timer utilization, reprogramming, sensor integration and memory management. With support for both IP networking achieved by the μIP stack [45] and low-power radio utilization with the Rime stack [47], a versatile communication subsystem is also part of the Contiki operating system. Similar to the TinyOS community, the Contiki developers are a very active research group, providing a number of additional tools available for download on their website.

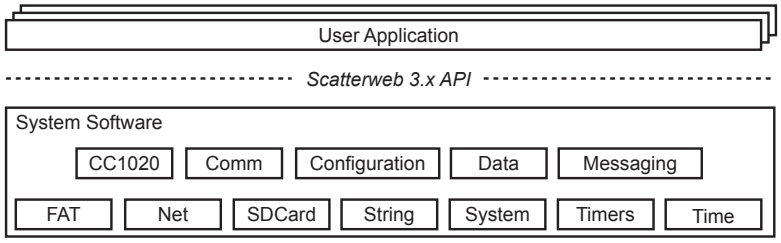


Figure 2.2: ScatterWeb 3.x software architecture

ScatterWeb firmware

Akin to the above presented full-fledged operating systems, the firmware of the ScatterWeb MSB430 also follows an event-centric approach for processing. Scheduling is performed within a super loop, polling individual modules that represent their hardware counterparts and dispatching incoming events to their designated handlers. Figure 2.2 depicts the available implementations that serve as abstractions for an application to rely on. Core functionality includes a module to control the microcontroller with support for basic tasks such as interrupt handling, a set of communication services such as packet handling and medium access and the provision of an interface to timer abstractions and I/O handling. Besides, a set of services that implement e.g. drivers for optional sensors or basic access mechanisms for secondary storage are provided as libraries, thus can be linked if required.

All in all, the system implementation is very lean, exporting a set of basic event handlers to be implemented according to application needs and an API restricted to the absolutely necessary functionality to a programmer. Since the ScatterWeb firmware, and likewise any application that builds upon it, is implemented in ANSI C, no explicit software development design is enforced, which is on the one hand beneficial as it leaves maximum control to the developer, but on the other hand is also a burden, as it requires a certain degree of expertise.

Considerable effort has been made to port e.g. a variety of MAC, routing and security protocols to the ScatterWeb platform, validating its versatile capabilities. Furthermore, ScatterWeb nodes have been utilized within a number of research projects and experiments with an impressive diversity of subjects, including concerns to autonomously position the sensor nodes [142, 64], provision of programming abstractions [7, 24] or utilization in hazardous environments [13]. A solid codebase is thus available for fast prototyping.

2.2 Event Processing Systems

In everyday life, asynchronous, ubiquitous incidents that have a meaning to their recipients and/or request some kind of reactions are commonly referred to as *events*. A traffic light turning to green, the crash of a hard disk of a computer or the birthday of a good friend are labeled to be events since they all share a sense of change: A significant modification in contrast to the previous situation of the person exposed to an event occurs and requires an approriate reaction. In the above mentioned context, a roaring engine of a vehicle, a replacement of the hard drive or a written birthday card may be observable consequences.

Likewise, significant change of (system) state is a central concern in computer science and can therefore be found throughout various domains: Operating systems feature interrupts and signals, the former to indicate a special event such as the expiration of a timer to the processor, the latter to allow for inter-process communication (IPC) of concurrent processes; Active databases utilize events to monitor relevant operations on their data, to incorporate integrity constraints, workflow management or alerting strategies; Events in programming languages may e.g. be used to model exceptional states of the running system or to comprise user and/or device interaction; Knowledge representation relies on a multitude of input events to derive complex states or high-level events, thus to infer knowledge from available data and build powerful expert systems. All of these areas model at least partially their input parameters determining program flow as events. Upon detection, these are filtered and processed by the invocation of corresponding actions or handlers.

Although the general idea of significant change constituting an event is widely accepted, the semantics of the term *event* as well as the actual event processing nevertheless varies among operational areas. Some treat events as transient, thus occurences have to be explicitly captured or will otherwise be lost, whereas for others, events are durative, thus may effect system state even at a later point in time when adding to a complex event. Furthermore, event handling or, more broadly speaking, the action space that is related to the occurrence and/or detection of an event, can range from side-effect free alerting to transactional processes.

After clarifying basic terms and pinpointing general implementation concerns in event-centric environments, we will briefly discuss the implications that the semantic differences in event and action space have on their design. Focussing on rule-based approaches to process events, we will provide a systematic analysis of popular rule dialects.

2.2.1 Historical background

Due to its versatility, no single, generally accepted definition of the term *event* can be postulated. The question how to describe, detect, process and react to events has seen a tremendous amount of research, primarily in two distinct areas - active databases and artificial intelligence [58]. As a result of separate goals

and efforts, both coined a slightly different notation and terminology for similar terms, a circumstance that in the current quest for standardization receives a lot of attention in the Semantic Web community. Considerable effort has been directed towards clarifying these differences and their implications to enable the fusion of both worlds [111].

Active databases

Starting in the late 1980s, the need for a better integration of monitoring capabilities for events led to the progression from traditional to active databases. Instead of purely reacting to user queries, inserts and updates, database systems were enhanced to concurrently inspect internal and external events, and to derive and recognize patterns of event combinations.

Active databases typically rely on the *ECA (Event-Condition-Action) paradigm* and corresponding event algebras to define complex event patterns and to enable the automatic triggering of actions in response to their detection. The structure of such event-condition-action (ECA) rules - "*ON event IF condition DO action*" - provides an elegant means to express event-driven behavior: Whenever a simple or complex event is detected, denoted in the *ON* clause, and the condition(s) in the *IF* clause evaluate to true, the action following a keyword resembling the *DO* part is triggered. Common events that are monitored with active rules include particular operations on the data itself, transaction events or method execution in object-oriented databases. Condition statements usually involve SQL queries with non-zero return values while invocation of SQL updates can be found in the action part. Well-known active database systems are e.g. HiPac [41], Starburst [152] or Sentinel [30] to name but a few.

Often times, the mere detection of simple events and a rigid, preprogrammed reaction executed by one dedicated event handler in direct response is however not satisfactory. Instead, particular patterns of event combinations allow for a better expression of application logic. The ability to detect composite events is a key feature of active databases and presumes the availability of a distinct model to describe such composition. Chosen operators for the constitution of a particular *event algebra* and supported primitive events differ however among implementations, see SAMOS [60] or COMPOSE [62] for details. A commonly cited event specification language is SNOOP [31]. Basic operators such as conjunction, disjunction and negation are supported as well as sequential composition, a bound selection and periodic and a-periodic event sequence definitions. With their help, the expressiveness of the event part of an ECA rule is enhanced and the description of high-level, application-specific situations with active rules is facilitated.

Knowledge Representation, KR event/action logics

The second area that has seen extensive research on event processing is *Knowledge Representation (KR)*. Here, the general idea is to capture causality, thus to develop

a set of axioms and notations to determine necessary and sufficient conditions for events to happen. From the fact that certain events are known to have occurred, KR approaches infer new knowledge on application-specific questions, system state or upcoming events. A prominent example for KR approaches are expert systems [25]: Knowledge bases are built from expertise knowledge in problem areas that feature a high complexity with respect to correlated data. Therefore, causality between events and actions or relationships between data items of interest is acquired and modeled using a KR formalism. Ideally, the obtained knowledge base can then be consulted in decision processes, e.g. for health diagnoses to correctly interpret patients' symptoms (MYCIN system [26]), in monitoring applications to derive critical situations (REACTOR [10]) or during planning (configuration of Vaxen computers [104]).

Rather than focussing on event detection, KR event/action semantics rely on event occurrence and inference mechanisms to reason about consequences. Albeit two methods, backward and forward chaining, for reasoning exist, inference for event processing typically favors forward chaining: Starting from available data, thus knowledge about already acquired and/or anticipated events, inference rules are applied to extract additional information and to determine proper consequences. The backward chaining approach is exactly contrary: Given a certain goal, the inference engine inspects available data and inference rules to find evidence that supports the hypothesis. Rules are analyzed by their consequents (THEN part) that match this hypothesis before evaluating their antecedents (IF part).

Similar to event algebras used in active database, the KR event/action logics build upon powerful formalism such as Event Calculus [92] or Interval Calculus [9] to allow for temporal reasoning. The former for instance divides the world in events that happen at a point in time and initate/terminate time intervals and *fluents*, which represent properties of the system that hold over time.

2.2.2 Terms and Definitions

Primitive Event:

- Within the active database community, the term event (a.k.a *raw*, *primitive* or *atomic* event) refers to an "instantaneous, atomic occurrence of interest at a point in time" [33]. Instances of primitive events can be detected by a system only if the event type or class is known a priori. Formally, an event (both atomic and complex) is defined as a function mapping from the time domain onto boolean values:

$$E : T \rightarrow \{True, False\}$$

$$E(t) = \begin{cases} T(rue), & \textit{if an event of type } E \textit{ occurs at time point } t \\ F(alse), & \textit{otherwise.} \end{cases}$$

Semantically, the focus of event processing in an active database sense is on event detection. The function $D(E, t)$ can therefore be introduced to denote that an event instance e of type E has been detected at time t. Events are consumed upon detection thus transient and do not contribute to future complex events unless explicitly bound via variables.

- The (temporal) KR event/action logics view on events is slightly different: Predominantly, events occur over an interval of time. However, since a discretized model of time is sufficient for most applications, an atomic event may be defined to happen at an atomic interval in time. According to this, event occurrence is generally denoted by the function $O(E, [t, t'])$. Hence, occurrence of an atomic event can be defined as $O(E, [t, t])$, a circumstance that is expressed e.g. in Event Calculus by the function "$happens(E, T)$". In contrast to active databases, events in KR are non-transient. Represented as durative facts, they can initiate and/or terminate validity intervals in which particular properties (fluents in Event Calculus) are true.

Table 2.2 summarizes the main differences of event properties in both worlds discussed above.

Complex Event: Complex events (a.k.a composite events) are built from occurred atomic or other complex event instances according to operators of an event algebra [111]. Note however that in order to reuse derived or detected complex events for subsequent detection of other complex events, these events have to be made persistent.

Event context: Conditions for event detection and/or deriviation comprise statements for analyzing and filtering the event itself, but often times also depend on the current context of the system, commonly denoted as the event context. Context characteristics may involve temporal aspects (such as whether another event has been detected within a given time frame prior to the current event), spatial (e.g. concerning the actual location of the event source), state (e.g. current power level of the processing entity) or semantic aspects (e.g. do the roles of event source and sink match) to decide on appropriate event processing.

Event Processing: The term event processing subsumes several steps to properly handle incoming events to a system. These include operations applied to (raw) event instance (detection, transformation, deletion), analysis of the current event context, eventually the selection and composition of events for complex event derivation and trigger of actions as a consequence of the overall detected situation. Event processing can either rely on a pull model in case the system actively acquires information about its environment, or on a push model when events are detected externally and pushed into the otherwise passive system.

Table 2.2: Event Processing Objectives

	Active DB	KR event/action logics
Event	instantaneous transient	interval durative
Concern	Event detection	Event occurence
Composition	Event algebra	Event calculus
Goal	Reaction	Reaction / Deduction

Several papers have pointed out that there are logical shortcomings and unintended operator semantics when reducing the temporal scope of events to points in time as promoted in early active database work [111, 58]. Briefly speaking, the problem that arises in this context is due to the combination of a volatile event definition and a lack of expressiveness for time intervals. The specification of e.g. a sequence of more than two events preserving event order is simply not possible in this case. Events can only be specified to happen after the detection of another, but not in between two events.

To overcome this deficiency, two different solutions can be applied. One possibility is to turn to an interval-based event definition, thus shift from a notion of event detection to event occurrence and adapt the utilized event algebra accordingly. The advantage of this approach is that it clearly maps the real-world model of complex events better: Instead of being detected at the time of detection of the last contributing event, the complex event itself is durative, spanning from initiation to termination event. However, this comes at the price of potentially complex interval conditions, which may affect both event definition and event processing in a negative manner.

A second solution is to keep the event history so that temporal reasoning, even with the limited event detection semantics, is possible. The drawback of strictly persistent events is obvious: The event processing system is prone to overflow, unless adequate heuristics to restrict kept events to only relevant ones are incorporated. On the other hand, the event model itself features less complexity, thus may be more accessible for certain application areas.

2.2.3 Implementing Event Processing Systems

The notion of events is an intuitive metaphor to capture reactivity to stimuli stemming from both outside and inside a system. Regarding the software design, event processing can be a quite complex concern that comprises event occurrence/detection and derivation of complex events via event fusion, context analysis and evaluation of time dependencies. The actual implementation of reactivity, or more precisely the integration of reactive behavior into the software stack, varies how-

ever among programming models and differs with respect to direct support of the above mentioned challenges.

The utilization of a certain programming model defines the conceptual view a programmer relies on to structure his programs. In case software development is e.g. exposed to an operating system abstraction, a programmer will most likely think in terms of processes, threads, semaphores and so forth. Given a programming language abstraction, language elements such as e.g. classes and objects or functions and variables will dominate the mental model associated with program design. The more expressive a provided abstraction is, the larger is usually the software stack needed to support it, a circumstance of great importance in the embedded domain. For instance, high-level programming abstractions seldom execute directly on the hardware of a system, but rather depend on a runtime environment which in return may be built upon code libraries, components of a middleware system and OS system calls. Taking these observations into account and relating them to event processing implementations, two main directions towards enabling an event model can be discriminated: process-oriented and event-driven approaches.

Note that in the following, we will solely introduce these two basic design strategies. A discussion of rule-based event processing, although being a subclass of event-driven approaches, will follow in greater detail in the subsequent section due to its importance in the context of this thesis.

Process-oriented Approaches

The first approach to object event processing is to incorporate the asynchrony of events into a sequential program flow. The most common way to accomplish this is to rely on multi-threaded programming and to encapsulate reactive behavior in individual threads. Whenever the occurrence of a particular event is recognized in a given context, the corresponding thread waiting for this event becomes active and can be executed by the operating system. Otherwise, the thread simply blocks, giving other threads the opportunity to get scheduled in the meantime. Typically, each event or set of related events is associated with one thread. Listing 2.1 visualizes a simple threaded program that processes events. The conditions for event detection/occurrence, often referred to as the *event header*, are associated with variables or devices monitored by the operating system which signals the thread scheduler to activate waiting threads upon change. Examples for event conditions include e.g. the reception of a certain packet, a timer interrupt or the manipulation of a specific state variable. Upon thread activation, the adequate *event handler* is called and executed.

Evidently, this approach is quite convienent from a programming perspective, since the control flow can be directly derived from the program text of each thread. Ordinary control sequences such as loops, conditional statements and function calls are at a programmer's disposal to express appropriate actions, essential to ensure reactivity of the program at any time, can easily be supported so that long-running

Listing 2.1: Event processing based on threads.

```
1  void main () {
2    create (thread_1);
3    // ...
4    create (thread_n);
5  }
6  void thread_1() {
7    event_t event;         //local variable for event data
8
9    wait (EVENT_1, event);     //blocking wait for event 1
10   handle (event.data);
11 }
12 // ...
13 void thread_n() {
14 // ...
15 }
```

operations or blocking calls do not corrupt program behavior. The burden of - possibly preemptive - scheduling of threads is left to the operating system, so that only synchronization and shared memory have to be mastered by the programmer.

On the downside, the utilization of threads introduces severe costs regarding stack memory. Each thread is provided with its own context and thread state which has to be pushed onto the stack upon switching the currently executed thread. This has of course a slightly negative effect the on the execution time as well, see [87] for details. Furthermore, threads do not provide support for detecting composite events at a conceptual model. In fact, the relationship between raw events and threads can be expressed as a one-to-one mapping.

Event-driven Approaches

In contrast to the first class, approaches that follow an *event-driven approach* allow events to explicitly control program flow. At no point in time, a program should block and wait for some specific event to happen. Instead, it is accepted that data flow does neither conform to a known schedule nor has a predefined order.

An unpretentious way to implement an event-driven system is to wrap all event handling into a single superloop as shown in Listing 2.2. After initialization, the system constantly polls for new events and invokes their event handlers if necessary. Due to its simplicity, this model is very lean, but offers almost no support for a programmer regarding e.g. event buffering and ordering, enforcement of timing constraints or efficient resource handling. Generally, since event-driven systems, especially in the embedded domain, often depend on to a single thread of control, invoked actions have to be non-blocking and terminate in a bounded amount time. In case actions contain blocking functions, these have to be transformed into two fractions: a non-blocking request and a completion event generated by an interrupt that can trigger the continuation of the prior action.

Listing 2.2: Implementation of a simple control loop.

```
1 void main () {
2   //...
3   while (true) {  //runs forever
4     if (event_1) then event_handler_1();
5     if (event_2) then event_handler_2();
6     //..
7     if (event_n) then event_handler_n();
8   }
9 }
```

A more sophisticated approach to put the event-driven idea into practice and expose a push semantic to the developer is a pattern sometimes referred to as the *event dispatcher pattern*. (Primitive) events emitted by event sources are pushed upon recognition into a global event queue. An event dispatcher launched at system start time (line 4) accesses this queue (line 2) to schedule appropriate handlers according to the event type of a dequeued event, see Listing 2.3 for an exemplary implementation. Therefore, event-driven programs consist of a set of actions specified as event handlers which are scheduled according to the order of their correspondent incoming events. Mapping of events to actions in this basic implementation once again adhere to a one-to-one mapping scheme, implemented via a `switch`-statement (lines 11-17).

Clearly, while event driven approaches mirror the asynchronous world of events better, they come at the cost of sacrificing the nowadays prevalent sequential programming model, and along with it the convenient programming construct of the call stack. Coordination, continuation and context preservation, things managed by the call stack in ordinary programs, now have to be addressed explicitly by the programmer. Management of system state and control flow internal to event handlers are exposed and request application-specific solutions. Valuable characteristics of event-driven design such as increased flexibility of programs and agility of systems and a more concise and cohesive model of the problem domain however add to the popularity of this model.

2.2.4 Rule-based Event Processing

Recall the definition of the term event processing in Section 2.2.2 which comprised more than mere detection of event occurrences and immediate action invocation. For the above mentioned models to conform to this definition, the one-to-one relationship of mapping events to handlers has to be substituted by a possibility to also process complex events. In this case, the relationship of events to handlers is broadened to a many-to-many mapping, with multiple (primitive) events contributing to different complex events. Multiple event occurrences, possibly in a given sequence, time frame and context are however difficult to express: Unless the number of complex events is deceivingly small and their nature simple so that

Listing 2.3: Common event-driven processing pattern.

```
1  void main () {
2    event_t event_queue[MAX_EVENTS];
3    // runs forever
4    event_dispatcher();
5  }
6  // schedule next element in event queue for processing
7  void schedule() {
8    event_t event;
9    event = dequeue (next);   //fetch next event from queue
10
11   switch (event.type) {
12     case type_1:
13       event_handler_1 (event.data);
14       break;
15     case type_2:
16   //...
17   }
18 }
19 void event_handler_1 (event.data) {
20   process (data);
21 }
```

encoding them directly with the help of state variables is an option, the usage of a higher-level of abstraction is favorable. Extracting significant features such as support for complex event specification which is common to a set of applications and shifting them to system responsibility not only leverages programming effort, but also reinforces code reliability, readability and maintenance. Popular concepts to handle this concern include frameworks that provide publish/subscribe mechanisms enabling exactly the circumstance described above of one event contributing to several complex events, or reactive rules that expose a language-based abstraction to complex event specification. Since the latter approach is examined in this thesis, we will provide an introductory summary on reactive rules.

Reactive rules, in contrast to logic rules, actively update and change system state upon execution. Due to their declarative nature and fine-grained modularity, rule languages have always been a popular approach to capture reactive behavior, with prominent rule engines being e.g. JESS [56] or commercial systems such as ILog [1]. Just as there have been different approaches towards the notion of events 2.2.1, processing languages based on rules as well differ in syntax, target domain and execution semantics. Two basic classes can be distinguished - languages that follow an ECA paradigm on the one hand and production rules on the other [19]. In both cases, the processing entity that interprets rules and schedules them for execution is referred to as the *rule engine*.

ECA rules

Event-Condition-Action rules consist of three different parts. The first part, typically marked by a keyword (e.g. *ON*), states the initiation event(s), the second denotes the condition part of the rule (started e.g. with *IF*) and a last (e.g. marked by *DO*) specifies the actions to be executed upon rule triggering. This clear separation into distinct parts can foster a separation of concerns which may in its extreme result in different sub-languages and even data models for each individual part of a rule: an event specification language for part one, e.g. SNOOP [31], a query language such as SQL for part two to query persistent data and a host language to specify actions. Action invocation includes raising new events, procedure calls, data manipulation and eventually modifications of the rule base. Strict separation has its advantages and shortcomings. A positive effect is that internal system state and data is not directly exposed to the event processing entity, a benefit that is of utmost importance in distributed settings. Then again, the flow of information between rule parts is not given. Due to the transient nature of events in ECA languages, event data has to be extracted and forwarded to subsequent rule statements, a circumstance usually solved via binding data of interest to variables.

The execution of a single rule is straight-forward: Upon event recognition and a positive condition evaluation result, the rules actions are executed in an atomic manner, thus either occur completely or none at all. In case multiple rules react to the same event, several scheduling strategies can be applied with different execution semantics. The scheduler can e.g. select a single rule from the so called *conflict set* based on a predefined priority ordering of rules, but also choose it in a non-deterministic fashion. Or, if by design of the rule base multiple matches can only occur by mistake, the scheduler may simply reject the complete conflict set and report an error. A last possibility is to select all rules in the set and execute them sequentially. However, it has to be clear what effect the generation of new events as a result of action execution has on following rules in this strategy - they can either be suspended being replaced by a new conflict set or events can be hold back until the complete conflict set has been executed. All mentioned cases are valid scheduling strategies, but of course have a big impact on the design space of the associated rule bases.

Although ECA rule languages originate from the Active Database domain, their popularity reached the general area of distributed systems long ago [32]. This is not a big surprise given the fact that events make a good abstraction to exchange information which has been extensively probed in Event Notification Systems (ENS) [28]. Furthermore, the push mentality of the event concept which is usually adopted nicely meets common requirements such as avoidance of unnecessary traffic, fast reactivity and low resource consumption. Nowadays, ECA rule variants are deployed as business rules, to coordinate workflows and especially in distributed web-based applications such as e-business and semantic web applications to provide required reactivity in a declarative manner.

Production rules

Production rule syntax differs from ECA rules in that instead of explicitly naming the event that activates rule execution, they react to changes in system state. Likewise, the syntax takes the form *WHEN* condition *DO* action, hiding the activation event. The term production (rule) itself originates from the Chomsky hierachy of formal grammar types [34], where rewriting rules are used to denote a generative grammar of a language.

Production rule systems have received a lot of attention e.g. in the area of Artificial Intelligence to serve as a format for knowledge representation, but have also been used to encode e.g. application logic or business rules. The number of rules is typically rather large, operating on a single working memory, a finite set of data items sometimes referred to as the *fact base*. The representation of data items is however left up to the data model. Unlike ECA rules, a system to process production rules shares this working memory among rules and rule parts, thus exhibits a tighter integration.

Regarding rule evaluation, production rule systems can in general follow both, the forward or the backward chaining approach to determine upcoming system state. However, if rules are applied in an event processing thus reactive context, forward chaining inference will be the algorithm of choice since it maps the data-driven incremental push semantics of incoming events: Whenever an update, due to an internal or external event, is recognized on the fact base, the precondition for rule execution is evaluated by means of pattern matching. Therefore, the situation described in the condition part is matched against the working memory which represents the current situation of the system. In case a fact or object of a certain type became available due to a preceding update or it diminished, or monitored values of fact/object variables changed, rule precondition and current state may match, leading the rule engine to fire the rule and schedule its actions. Statement execution will be invoked on data items matching the condition part, also called the *rule instance*. The *refraction principle*, a fundamental concept in production rule systems to prevent infinite triggering of the same rule, denotes that once a rule instance has been executed, its condition has to become false on these data items.

Execution semantics of multiple rules reacting to change has to be specified exactly in order to guarantee intended rule program behavior. The main criterion to differentiate production rule execution algorithms is whether the execution of a particular rule from the conflict set influences the successive conflict set (and to what extent) or not. In stateful implementations, the eligibility of rules to trigger varies during conflict set execution, thus the state reached after one rule has been processed may validate previously inactive rules to join the conflict set and remove rule instances whose condition parts become false. Simply speaking, changes on the working memory directly effect rule eligibility. The Rete Algorithm is the most well-known pattern matching algorithm to efficiently implement stateful production rule systems [52]. It exploits the fact that updates to the working

memory usually influence only a small number of data items and that patterns in the condition part tend to occur in more than one rule to minimize the search space for conflict set recalculation. A stateless implementation of a production rule system is applied in case attributes denoted in the condition part of a rule cannot be modified by executing a rule's action. Typical application areas that lack the need for inference chaining, which thus can be implemented in a stateless way are e.g. filtering applications that operate on a tuple-by-tuple basis or data validation methods.

Production rule systems are often applied to problems that are logically rich and focus on state management. Especially stateful rule engine implementations with their inference chaining capabilities offer a tool to process event sequences and provide case-sensitive reactions dependent on the particular system state.

2.3 Concluding Remarks

Wireless sensor networks have now been explored, deployed, tested, adapted and studied for almost a decade in a variety of applications, revealing their great potential, however at the same time the inherent properties requiring utmost attention when yielding a successful utilization. Many of these, e.g. the demand for unattended operation, physical robustness of nodes, conceptual robustness to cope with unreliable links and high network dynamics, or the concern of node heterogeneity, have already become apparent in the early years [124]. As real-world experiences have been scarce at that time, these challenges have primarily been derived from envisioned application scenarios and issues that represented interesting research questions in this context. Correspondingly, primary concerns also included extremes such as excessive miniaturization or very large network sizes, which have lost their appeal over time. Back then, the focus has definitely been on the big, visionary picture.

Reality struck most early projects when simply a lack of solid software development and extensive testing in combination with an overly-ambitious time schedule resulted in, at least partially, non-operational deployments. Identifying those circumstances that rendered achieving project goals difficult triggered a shift of perspective to node-local concerns. Since a stable execution of software is mandatory to obtain practically usable networks, a great deal of low-level software for sensor nodes and tools aiding in the software development, debugging and evaluation process has ever since been implemented. Furthermore, deployments revealed that although the design space for wireless sensor networking applications is huge, each individual project typically has to be only concerned with a small subset of challenges to be met, emphasizing the need for careful, modular and very targeted software design. Thanks to implementations such as the presented operating systems, both platforms and software stack have nowadays matured, allowing computer scientists to utilize WSNs at quite reasonable effort. Bridging the gap to domain-experts to really make WSN capabilities available for a greater

audience without limiting these networks per se to a set of toy instructions is definitely a very urgent topic to address.

Due to the primarily reactive nature of most applications for wireless sensor networks, the quest for a small footprint and energy-aware implementation, event-centric processing is the de facto standard software design for crafting the protocol stack of WSNs. Therefore, the second part of this chapter has been concerned with clarifying how the term *event* has been conceived over time and application domains, and what effect a different concept has on the actual implementation of event processing systems. It is interesting here to point out that especially high-level abstractions such as rule-based programming paradigms reveal severe semantic differences which can be appointed to their sources of origin. As a consequence, a concise specification of model semantics has to be provided when introducing or utilizing such a model in order to prevent a false interpretation, a concern that will be discussed in Chapter 5.2.

Chapter 3

Related Work

Support for rapid development, yet dependable execution of software for wireless sensor networks is not only mandatory to enable their adoption beyond academia, but also to provide a solid ground for fundamental research on algorithms. Problems such as for instance scalability, in-network coordination, efficient data processing or real-time, distributed event detection in a massively distributed environment have to be studied and solved for these networks to be used to their full potential. The key to meeting the challenges introduced in Section 2.1 is to provide a sound level of abstraction that software development can rely on.

Simplification of non-trivial control or data aspects greatly eases application development and is therefore a central method in computer science to make complex systems accessible and controllable. Generally speaking, abstraction allows a programmer to concentrate on a few, basic concepts rather than having to deal with a mass of details. WSNs comprise a mass of details concerning very different problem areas, but at the same time offer very little processing capabilities on the nodes themselves. Hence, proposed abstractions for this domain differ in concept, mechanism and practice they implement, thus problem they solve and viewpoint they provide.

This chapter will give an overview to what ends abstractions have been explored and what functionality for application development is explicitly supported. Therefore, a description of basic classes of abstractions that share a common idea will be first introduced and analyzed. Dependent on the provided classification, examples of work related to this thesis will be presented in a second step and qualitatively analyzed in a third.

3.1 Perspectives on Abstraction

Abstraction provision for wireless sensor networks can be as diverse as distinct challenges have to be addressed. To narrow down the scope of abstraction mechanisms reviewed in the following, we assume a basic operating system to be present, thus low-level hardware related functionality relevant to upper layers to be implemented and accessible via a specific interface. Therefore, the software layer in between system and application, which we denote as the *middleware* layer, and its characteristics are subject to analysis.

The term middleware is traditionally associated with hiding the protocol layer, sophisticated service provision beyond OS features and deployment in a distributed environment. Bernstein [18] specifies a set of criteria commonly addressed by middleware services including independence of the chosen platform so that middleware services have to be portable to a variety of system architectures with modest and predictable effort, supply of functionality that meets the need of a wide range of different applications and distribution of the service itself. In short, the compilation of a catalogue of such properties enables a *functional* view upon middleware implementations.

In a wireless sensor network context however, the application of above mentioned criteria to classify approaches is not conclusive: Due to the limited resources available on sensor nodes, middleware development is much more driven by classes of applications to be supported rather than aiming at a general-purpose solution. Suggestions for classification therefore explore middleware approaches from a variety of angles. Key to a classification presented by Heinzelman et al. [74] is e.g. a distinction between providing a reactive or a proactive handle to the sensor network by means of a middleware. This choice contributes to the fact that application information is usually not strictly isolated from underlying software to enable better resource utilization [124], either to support QoS-aware applications [43] with proactive measures or to enable application-specific reactions to network dynamics. Adaptive fidelity algorithms [157], cross-layer optimization [102] and QoS specification methods have therefore been studied thoroughly. Overall, one can deduce that the perspective on abstraction taken here is to abstract from the *network*.

Analyzing the *programming paradigm* that a middleware offers to fill in the gap between system and application entails a different view on sensor network middleware [121]. The distinctive measure then is the mental model that a programmer may rely on during the software development process. A taxonomy of programming models is presented in [115], dividing approaches into those that offer some kind of support (either for composition, to overcome distribution or optimization issues) and another group that enables a programmer either to abstract from node-local or global node behavior within the sensor network context, with the latter being refined in [133]. While the commonality all approaches share is supplying a dedicated way of programming to task sensor networks, the intention what problem to address with this differs among approaches. Generally,

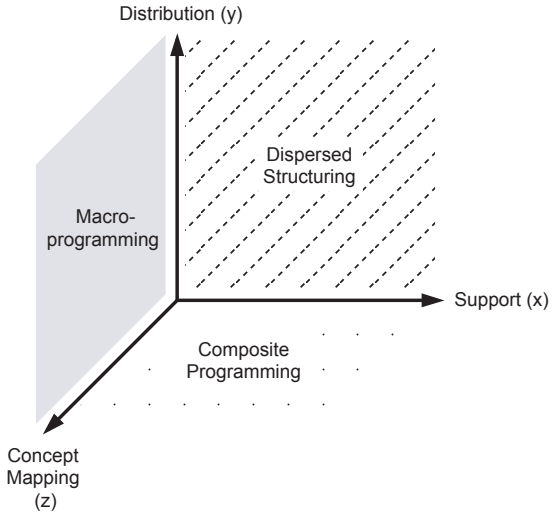


Figure 3.1: Classification of middleware abstractions for wireless sensor networks

when adopting a certain programming model, the level of abstraction from control and/or data concerns is increased.

In order to provide a more holistic view on sensor network middleware, we present a classification that combines the findings mentioned above into a single model depicted in Figure 3.1. While lacking a degree of detail in terms of functional correlation between middleware and application it supports, it allows for quickly grasping motivation and method of proposed approaches.

Each axis represents a basic dimension of abstraction that addresses a core problem area a developer is exposed to when dealing with wireless sensor networks. *Support*, assigned to the x-axis of our model, can be attributed to those approaches that implement supportive measures in terms of services or components to solve a specific problem or to fulfill a certain task. Encapsulation of predominant WSN-specific needs into self-sufficient components, an orientation towards light-weight middleware functionality enabling context-dependent linking of software, modular instead of monolithic software architectures and the urge for re-applicability reflect prevailing design rationals. Characteristic approaches therefore usually provide a predefined, architectural framework which will be reviewed in Section 3.6.

On the y-axis, we denote *distribution* as a major challenge thus fundamental dimension for abstraction provision. Here, coordination and control of network subsets or *groups* is the prime concern. The granularity at which such subsets can be formed and addressed varies among middleware implementations and may span from nodes within a one-hop network vicinity, often referred to as a *neighborhood*, over spatial or logical groups to complete network coverage. As a consequence of utilizing a wireless medium to interconnect a possibly large number of sensor

nodes, additional complexity arises to encounter network dynamics. A selection of approaches classified to explore this abstraction dimension will be presented in Section 3.2.

To overcome the discrepancy between system- and problem-oriented viewpoint that the sensor network domain imposes on application development, a common mechanism is to supply a dedicated *programming paradigm*. Therefore, the z-axis of the classification is associated with what we call *concept mapping* as a way of reflecting the ability of approaches to conceptually meet key WSN challenges. Due to the embedded nature of deployed devices, the syntax of utilized languages is often ill-suited to express application semantics, incorporation of system-related functionality has to be addressed and thus straight-forward application development is hindered. Section 3.4 points out relevant projects relying on a concept mapping abstraction and analyzes their advantages and shortcomings.

Naturally, many approaches exploit abstraction in more than one of the presented dimensions. Moreover, classifying a particular approach is neither necessarily a matter of a boolean decision for one class but can be gradual, nor is it a significant alleviation for middleware selection if left too vague: Nodal distribution can e.g. be objected at different levels of abstraction, ranging from node-level to network-level transparency. Likewise, a countermeasure can be implemented in a variety of ways, utilizing for instance distributed data structures, language supplements or predefined protocols. Both, concept and implementation, can have a serious impact on selection preferences. To achieve a finer categorization of available middleware implementations, we therefore subsume approaches that substantially interleave two of the discussed dimensions under their own label. Hence, approaches entitled to belong to the *macroprogramming* plane, see also Section 3.3, introduce a programming model particularly to overcome distribution challenges. In case modularity has been an integral design rationale of a domain-specific programming abstraction, we will refer to corresponding implementations as representatives of *composite programming*, being reviewed in Section 3.5. Featured middleware abstractions of the (x,y)-plane, consolidating supportive functionality with distribution transparency, are discussed in Section 3.7 and will be denoted as *dispersed structuring* in the following.

At this point, it is noteworthy to mention that the classification as such does not yield a qualitative evaluation of approaches in terms of individual practicality: A higher level of abstraction or integration does not automatically correspond to being the better platform. Instead of providing a sharp functional or analytical distinction between middleware approaches, the intention of this classification is rather to point out the general design space of application development on top of selected approaches presented in the following. At the end, a summary of the findings is compiled in Table 3.1.

3.2 Distribution Abstractions

As soon as sets of nodes have to interact to accomplish a task together, the programmer has to select, task and organize the corresponding nodes accordingly. Middleware approaches that implement a distribution abstraction offer support on at least one of these and allow to abstract from problems arising from nodal distribution.

An early representative called *Hood* has been proposed by Whitehouse et al. in [150]. This middleware architecture provides an interface for an application to a subset of the sensor nodes that are in physical vicinity, called a neighborhood. Based on criteria for choosing neighbors and a set of attributes to be shared, a user can specify different kinds of neighborhoods. Hence a neighborhood is formed by those nodes within a one-hop distance that are e.g. able to provide temperature readings. Hood then handles any management issues arising, like supervision of neighborhood lists, data caching and sharing among nodes and the definition of messaging protocols.

Communication within a neighborhood is based on a broadcast/filter mechanism, thus follows a one-to-many pattern. If a node wants to share one of its attributes, it simply broadcasts the value. Incoming packets are filtered, and nodes can determine whether or not the received attribute should be cached. Sharing core ideas with reflective memory, each node will allocate a mirror structure for reflecting values of every node in its neighborhood list. There is no feedback to the node that sent the value, so in contrast to concepts building upon reliable networking, Hood can cope with asymmetric links between nodes but at the same time cannot guarantee message delivery.

Looking at the programming part of the approach, a neighborhood becomes a programming primitive. To create a new neighborhood and to allow its individual parameterization, a code generation tool has to be invoked by the developer. The system itself builds upon TinyOS and uses its core communication components, see also 2.1.3. Interfaces offered by Hood provide handles to access neighborhood attributes and define sharing and updating strategies. Furthermore, values of neighbors stored locally on a node and can be annotated with so called scribbles. These simply note extra information, for example the quality of the link to the mirrored node.

Overall, a programmer who builds applications upon Hood is given the possibility to address and control functional parts of a network vicinity together, instead of issuing single nodes. Thus, this project alleviates effort for maintenance and takes the burden of dealing with distribution of updates in a neighborhood from the programmer. On the downside, Hood only provides best-effort message delivery based on broadcasting and restricts neighborhoods to one-hop. Consequently, the usage of Hood makes only sense in environments that employ cheap broadcast mechanisms and not e.g. rely on TMDA-based MAC protocols. Furthermore, Hood is not a runtime environment: dealing with race conditions, thrashing or live locking due to miscalculation in memory consumption are not addressed.

Logical Neighborhoods, proposed by Mottola et al. in [105] expand the idea of forming neighborhoods from a set of nodes in a one-hop vicinity as proposed by Hood to a logical partition of a sensor network that operates in a multi-hop manner. To accomplish this task, the Logical Neighborhoods framework provides two basic components.

First of all, a declarative language called SPIDEY is utilized to specify a node's exported attributes in a template, and to define neighborhoods with the help of predicates over such node templates. The actual membership of nodes to a specific neighborhood is determined at instantiation time, and requires an application programmer to declare the starting point of neighborhood construction. A nice feature to bound energy consumption on sensor nodes in terms of messages sent is that the framework implements a credit-based cost function, which may be used to restrict the scope of neighborhoods, and thus enables application-level control of resource consumption.

The second part of the Logical Neighborhood's framework is a neighborhood routing primitive that enables nodes to multicast values to other all members of the neighborhood. To this end, a structureless routing mechanism has been implemented that enables message delivery even in dynamic environments.

From an application programmers point of view, distributed applications may be built in a cost-sensitive manner without having to explicitly develop underlying network protocols when relying on Logical Neighborhoods. Nevertheless, node-level code has still to be written to express application semantics such as data filtering, processing and forwarding. Additions to Logical Neighborhoods have been proposed to furthermore raise the level of abstraction [112], but will be omitted here for brevity.

A third approach explicitly using the notion of a group as a primitive to specify data sharing mechanisms is *Abstract Regions* [148]. Dependent on spatial properties such as geographical location or radio connectivity, regions of sensor nodes can interact, thus share data in a <key, value> manner, establish shared variable values within a region or count participating nodes. Implemented on top of TinyOS, it exports an interface for the above mentioned functionality, but leaves details on region definition up to the programmer. To cope with the problem of hidden communication cost that transparent networking bears, the authors provide a tuning interface for applications to specify bounds on e.g. the number of message retransmissions.

The implementation of Abstract Regions seems to be neither fish nor fowl: on the one hand, complex operations e.g. reduction of distributed values into a shared variable have been implemented and provided while on the other, it is left completely open in how far means for region specification are available. Furthermore, the span of abstractions within the framework introduced ranges from very high when looking at region operations to MAC-layer concerns to be tweaked by the application.

3.3 Macroprogramming Abstractions

Distribution abstractions provide an interface that offers primitives for smart information diffusion to a subset of nodes in the network, implemented in the host language and accessible usually with the help of an API. In contrast, approaches we consider to be macroprogramming approaches make substantial use of a programming language itself being a tool to address distribution. In a sense, we adopt the term macroprogramming as it is generally referred to programming abstractions for specifying global, network behavior in the community [115], which introduces a shift in perspective from tasking individual nodes to writing network programs.

TinyDB [99] and *Cougar* [158] have been early projects to alleviate network-level programming from application programmers. Their idea of a middleware is using a database abstraction to enable the utilization of a well-known, declarative programming language upon the distributed nodes without an application programmer having to consider any network issues. Therefore, the network established by the sensor nodes is understood as a distributed database which can be queried using a query language. Since both approaches share general language semantics, we will focus on TinyDB in the following.

Besides basic SQL primitives, the authors of TinyDB added some essential keywords specific to the sensor network domain to enhance the language. Each node of a TinyDB network contributes one row to a single, virtual table to query execution, and each column represents one of the attributes that can be queried. A query processor is running on every node to handle and to possibly aggregate the sensor data questioned by the query specification. Thus to obtain values from the network, a user issues a query, which is then automatically routed to all nodes. TinyDB maintains a spanning tree from the node where the request has been initialized, so that resulting data can be sent back in reverse direction. Queries may be marked to be evaluated periodically, or values to be aggregated, summed up or grouped on their way back through the network. Any maintenance concerning bootstrapping or failure of nodes or routing issues will be handled by TinyDB without any interaction with the programmer.

With its SQL-style programming manner, TinyDB offers a high-level interface to a sensor network that is already widely accepted. On the downside, querying approaches do provide only side-effect free interaction, a fact that restricts application development substantially. Furthermore, tasking beyond a central-sink architecture for data streaming is not intended.

The *Regiment* system as proposed in [109] and its actual implementation evaluated in [108] supports application programmers by providing a functional domain-specific language. Instead of specifying node-level behavior, Regiment offers macroprogramming primitives and thus enables the specification of global programs that are compiled into an intermediate language to be interpreted on individual nodes.

From the authors point of view, the core element that wireless sensor network application scenarios feature are data streams. Originating from a set of spatially distributed sensor nodes, Regiment therefore provides language primitives to process these streams, to transparently aggregate data from multiple nodes forming neighborhoods and to abstract from data acquisition details and storage. The result of processed data streams, which can for example be detected events such as a sensor value exceeding a threshold within a certain region of nodes, are automatically forwarded to a predefined base station.

Regiment programs are composed of functions that manipulate so called signals, which represent streams of data samples at a given time, and of functions that can act upon regions, which represent collections of signals. To construct a region, a programmer may either form it starting at a certain node in the network via invoking a spanning tree algorithm, or he can rely on two different gossip-based primitives. With the help of these constructs, programs that include filters on data streams of nodes, their organization into neighborhoods and their automatic delivery at a central entity can be easily defined. Note that it is not possible to alter node-local state since the Regiment language is, just as TinyDB, side-effect free. Source code is compiled to an intermediate language called Token Machine Language (TML), which is interpreted on the nodes accordingly. A token is akin to an active message, encapsulating a payload of private data and triggering the execution of an associated token handler upon reception.

The main benefit of utilizing Regiment is that it offers a very high level of abstraction to organize data streams, which makes it especially appealing for rapid prototyping. Once again, Regiment is side-effect free, limiting applications to mere read access.

Kairos [115] and its successor *Pleiades* [91] share the same idea of annotating sequential code with macroprogramming statements. They both provide language primitives to access node-local state and iterate over a set of nodes, but differ in the way they support serializability, concurrency and code migration.

Three simple extensions to C sourcecode have been introduced by Kairos: The *node* data type allows for logical naming of sensor nodes and exports a set of common operations on nodes, a call of *get_neighbors()* returns a list of one-hop neighbors that may be manipulated iteratively and the ability to access data remotely is supplied for named nodes. To put into effect the annotations mentioned above, a preprocessor filters the program for these enhancements. A compiler then generates node-level code, with Kairos commands being translated into calls to a Kairos runtime that has to be preinstalled on every node. Any variable that is subject to remote access, so called *managed objects*, or referenced by a node but resides at a remote node, so called *cached objects* are managed by the runtime environment. Program execution that involves remote access follows a synchronous execution model and is internally dispatched to asynchronous message passing between participating sensor nodes by the Kairos runtime.

Pleiades basically explores a similar approach, thus augments C with language

primitives, but furthermore offers support for reliable concurrent execution of code on multiple sensor nodes. The Pleiades runtime takes care of synchronized access to shared variables, guarantees serializability and locks resources accordingly. Both, the compiler and the runtime system support a distributed deadlock detection and recovery algorithm to avoid potential deadlocks when invoking a concurrent iteration over a set of nodes, a feature made available by introducing a so called *cfor loop*. Furthermore, the program will automatically be partitioned by the compiler into nodecuts, node-level programs, whose control flow is managed and migrated between nodes by the runtime environment in order to minimize communication costs. Pleiades programs are translated into nesC code, with the Pleiades runtime being a collection of TinyOS modules.

By injecting new statements that guard shared variables and allow for simple manipulation of a set of neighbors, a programmer relying on these macroprogramming approaches is relieved from having to explicitly address shared state. This very convenient feature naturally comes at the cost of unknown and uncontrollable costs for message transmission.

An approach to offer a configuration environment for nodes in sensor networks is suggested by Frank et al. [123] [55] with the *Generic Role Assignment* project. Motivated by the challenge of intricate configuration issues of large scale sensor networks after the deployment of nodes, their goal is to enable network self-organization: nodes have to evaluate their own status within the network, compare it to surrounding nodes and then adjust their behavior themselves. To accomplish such automated, self-sustained operation, nodes agree on specific roles each of them will take within the network that have to be specified prior to their deployment. A role is denoted with a set of predicates over system properties of sensor nodes. Conditions of such rules to adapt a role may involve local information, but also span over a well-defined set of neighboring nodes properties. Language-wise, simple boolean predicates, a conditional, distributed count and so called retrieve predicates to bind results of distributed evaluations to a local variable are available. After diffusion of a compiled role specification into the network, the role evaluation process will start automatically and eventually end in a stable network configuration.

Generic role assignment acts as a decentralized framework organizing the distribution, communication and evaluation mechanisms involved in role constitution. This is especially beneficial for applications that can be divided into preferably small number of distinct roles a node can adapt. A threat to energy-efficiency may be possible oscillation of roles: minor local changes can eventually trigger a global, network-wide reconfiguration process. A major disadvantage is the lack of ability to integrate node state into role assessment which prevents fine-grained, state-based programming of sensor networks.

3.4 Abstraction via Concept Mapping

Pure domain-specific languages that neither fall into the category of languages or language additions promoting network-level programming nor foster a modular, component-based approach, but provide substantial abstraction from sensor node hardware and programming style are subsumed under the label of concept mapping approaches. Embeddedness, event-centricity and stateful coordination are the main challenges investigated by research in this domain.

A prominent way of supporting sensor network application development is the provision of a runtime environment or virtual machine. A multitude of efforts have been put into practice with e.g. Squawk [130] running Java ME CLDC 1.1 compliant programs, Scylla [131] designed to enforce reliable error recovery and memory access and VM* [90], a framework for synthesizing individual, application-specific runtime environments interpreting Java code. To this end, Lewis et al. [93] have also developed a family of virtual machines, subsumed under the keyword *Maté*, as well as a specific bytecode they can interpret to tackle the problem of retasking a network at runtime and provide a concise domain-specific language for sensor networks. The vanilla VM, *Bombilla*, executes statements written in *TinyScript*, a simple imperative language resembling BASIC syntax, while in addition *Mottle*, a scheme-inspired C derivative is available supporting a richer data model. Both languages are compiled down to an assembler-like instruction set that combines low and high-level instructions, and allows three possible operand types to be used: values, sensor readings and messages. Besides basic instructions for arithmetic computations, halting and branches, sensor network specific commands are available which will in turn be issued on a stack-based architecture built upon TinyOS. A thread pool of *contexts* encapsulate reactions to system events and application commands, own an individual operand stack and can be scheduled concurrently. A build-in routing algorithm may be called by issuing a single instruction, which is in charge of sending the specified packet to its destination. Also, another specific instruction allows packets to forward themselves and install new applications in the network that the packet encapsulates, thus realizing viral code updates. Eight instructions are left undefined for application-specific implementation to provide a tailored language specification, see [95].

A safe execution environment as provided by a virtual machine hides the complexity of the hardware or, in this case, TinyOS's complex, asynchronous execution model, and prevents system crashes. The instruction set design is especially targeted to the sensor network domain. On the downside, the stack-based architecture and the specified ISA are at a very low-level of abstraction and intentionally lack expressiveness: Programmer intervention is part of the design strategy.

Support for event-centric programming and stateful coordination of actions has been proposed by Kasten et al. [88] with the *OSM (Object State Model)* programming model. Applications expressed in OSM have to be partitioned into the set of

states they convey and events and/or state changes that lead to state transition. Resulting state machines may be hierarchically composed into superstates, as well as declared to run (conceptually) concurrently. OSM applications are compiled into native C using Estrel as an intermediate language.
The main benefit of utilizing OSM is its support for explicit state manipulation thus flow control and transparent scoping including memory management for variables declared to deal with application state. The authors thereby object to the problem of sharing state among different actions in event-centric programming: since local variables are automatically released from the stack after the respective function has run to completion, state may either be preserved via global variables at the cost of constant memory locking or via manually crafted state structures often prevented due to lack of dynamic memory management upon sensor nodes.

Although OSM empowers developers with a well-defined interface to stateful coordination, the language semantics fall short in respect to expressing events. Naturally, the occurrence of multiple concurrent events within one state can only lead to one state transition. Events are thus consumed on a priority-based semantic. As a consequence, the ability to specify a concatenation of multiple events necessary for state transition is not part of the OSM features.

A combination of state-based shared memory and a rule-based behavior definition for sensor nodes is pursued with the *Dynamic Embedded Sensing and Actuation Language (DESAL)* [12]. Programmers are equipped with a possibility to specify directional shared variables to serve as a communication substitute between program components. To utilize the soft state shared memory provided by DESAL, a developer denotes, possibly conditional, read or write bindings between variables which are then governed at runtime by a rule engine. Due to the unreliable wireless medium, distant variables may not yet be bound at runtime when requested to be evaluated, a problem that DESAL addresses by skipping the associated commands. Application semantics are mapped to rule specification, which take the form of a list of boolean expressions over state variables that guard state modification. Rule evaluation is triggered on a periodical basis, individually denoted with each rule body and scheduled in a best-effort manner.

Handling distributed state across unreliable networks, as well as addressing event-centric data processing are challenges that certainly need to be addressed. While the solution proposed with DESAL intends to overcome inconveniences caused, it runs short in several aspects: First of all, the specification of bindings assumes prior knowledge of the network setup to run in a controllable manner, a circumstance that limits applicability of DESAL. Although dynamic binding of variables is available, the entire program semantic may change if utilized leading to adaptivity that is rather harm- than helpful. Furthermore, reaction to events is artificially delayed by the scheduling policy for rules that questions a developer to explicitly time the evaluation which may not scale with increasing application complexity.

A quite similar rule-based approach to facilitate application development has been introduced in [128]. Once again, declarative rules are embedded into imperative control sequences, a design choice which aims at masking the event-loop without having to sacrifice rule-based application logic. Here, sets of rules combined into tasks, which are evaluated together every given time interval, encapsulate control flow. Once again, the authors chose static over dynamic scheduling motivated by the fact that this allows for compile-time program verification permitted due to predictable scheduling. This way, runtime reliability can be assured for a given network topology within a synchronized network. Interaction between nodes, rules or with the underlying system for sensing and actuation is modeled with the definition of abstract channels, which in turn are implemented via declaration of persistent memory on a node's heap. Channels may either be accessed with a send or a receive operation, thus share directional semantics with DESAL but allow for multiple readers at a time. For maximum reachable energy efficiency, the compiler should be aware of the underlying TDMA MAC protocol to schedule energy intense tasks such as logging during the awake time of a sensor node, thus maximize sleep cycles.

Presented language primitives in the paper share a high level of abstraction, but lack any pointers towards a real-world implementation. Therefore, it is questionable in how far the ideas presented are really tested within a testbed.

3.5 Composite Programming

Difficulties in application development in an embedded context are often a result of a low level of abstraction concerning the underlying hardware. Domain-specific languages that are explicitly designed to enable modular composition of middleware functions can be classified under the term of Composite Programming approaches. The discussion of representative approaches is subject to this section.

With the prime intention to serve as an intermediate language that higher-level abstractions (such as Regiment) can compile down to, Newton et al. [107] propose the *Token Machine Language (TML)*. The general model evolves around the notion of a distributed token machine running on the sensor nodes, scheduling individual tokens via their associated handlers for execution, thus borrowing the idea of vertical integration of execution and communication from Active Messages [144]. Each token possesses private data to carry its state and can be augmented with a set of arguments when traveling through the network or during local interaction to allow for parameter passing. All token objects are stored locally on a node within the sole dynamically allocated memory, a heap called token store, with their private memory only accessible by their dedicated handler. Interaction of token handlers with the token scheduler and the token store is restricted to a set of predefined operations to schedule, query, distribute and delete tokens, while manipulation of private token data or the bounded memory available for shared

data is left unrestricted. To avoid the introduction of a dynamically growing call stack and to allow for guarantees on termination of handlers, the execution of subroutine calls is split at compile time using CPS (continuation passing style) transformations into pre- and postcall handlers. All live data is explicitly pushed into a subtoken used later on for continuation. Non-recursive subcalls can also be inlined if the preset upper bound for maximal execution time per handler is still met. Nevertheless, the implementation of TML cannot support real-time scheduling due to the underlying TinyOS operating system and its split-phase operations e.g. utilized for sampling a sensor: events firing within the operating system introduce unpredictable time delays resulting in best-effort scheduling of tokens.

To enable support for a variety of applications, application-specific language additions have to be integrated into the interface offered to token handlers, a circumstance that reveals TML to rather be a model for component-based application implementation than a fixed language. Also, since token interaction is not directly allowed within the handlers it is evident that any coordination has to be pushed to shared memory. It is left unclear in how far race conditions are objected. Regardless of the difficulties mentioned above, application implementation in TML offers abstracting from memory management when adhering to its premises. In its current version, TML is build on top of TinyOS with its runtime as well as a TML program running as TinyOS modules after compilation. A lean implementation comes at the cost of direct linking of TML programs to the operating system.

With the introduction of the *sensor network application kit (SNACK)*, Greenstein et al [67] seek to provide a modular, component-based service library accessible to application programmers for sensor network tasking. To achieve this, they introduces a service-specification language operating on top of nesC [61] and especially designed to enable controlled sharing of variables as well as parameter passing between components. A compiler will parse SNACK-language service compositions and translate given component interdependencies into nesC component source code, which can then eventually be compiled to a binary image. A library, being the integral part of SNACK, consisting of popular services such as messaging, storage and a timer abstraction outsource low-level system interaction from application semantics when linked to the application.

SNACKs component-based specification language is a nice way of introducing modularity into nesC programming, with valuable services already provided as a library by SNACK developers. Apart from this, component implementation does not provide any extra benefit from utilizing pure nesC code: split-phase execution, concurrency and interrupt handling do still have to be understood and addressed, and another high-level configuration language learned. Therefore, using SNACK makes sense in case a viable library is already at hand.

3.6 Abstraction via Support

In contrast to the other primary dimensions, approaches that offer abstraction via support have no predefined functional or conceptual mechanisms they convey, but rather provide an infrastructural framework to encapsule functional units and mediate between application and system or system-related functionality. Representative approaches provide network management services such as e.g. code distribution [82] and take the form of libraries [35] or frameworks [102].

RUNES [37] is an approach that fulfills the classical requirements of a middleware. Designed to alleviate problems arising from heterogeneity of interacting devices, both in terms of manufacturer, operating system and system capabilities, and dynamic network settings, the authors propose a supporting middleware. Self-contained components feature necessary middleware functionality and can be individually deployed at runtime according to application needs. To enable this, a component model serves as a basis to specify basic runtime units and their corresponding interfaces in a language-independent manner. A middleware kernel, written in the host language of the device and running on each participating node in the network, processes modeled interdependencies of components at runtime.

A variety of platform-dependent implementations of the component model have been developed, including a Java virtual machine based implementation, a C/Unix-based implementation as well as an implementation running on the Contiki operating system to ensure the applicability of the model on heterogeneous systems. Components available for reuse include a data dissemination and a data acquisition component for sensor nodes as well as a data logging and packet forwarding component for desktop computers.

The clear separation of platform-dependent and middleware concerns provides an application programmer with a nice tool to develop dedicated, clean components. Nevertheless, low-level details of embedded programming are still exposed to a programmer since RUNES offers a tool to design programs rather than a mechanism to explicitly address inconveniences. A change in perspective from offering functional to conceptual support can be observed.

Impala [97], is an architecture implemented within the ZebraNet project [85]. The primary design goal has been to build a modular, lightweight runtime environment for applications that manages both devices and events. Hence, Impala splits the field of duty into two layers, one to encapsulate the application protocols and programs for ZebraNet, and an underlying layer that contains functions for application updates, adaptation and event filtering. Application programming follows an event-based programming paradigm, thus any application deployed upon the nodes has to implement a set of event and data handlers to respond to different types of events, including timer, packet, data and device events. Besides supplying event filter mechanisms, Impala emphasizes the need for integrating adaptation and updates of applications at runtime within the system architecture.

Adaptation of an application or an application-level protocol can become necessary due to changes of the system, e.g. failure of certain sensors or low battery level, as well as application specific modifications, e.g. a sudden drop of successfully delivered packages. A middleware agent, the Application Adapter, checks the overall state of the system on a regular basis and selects the most suitable configuration according to the present circumstances. Dynamic software updates may be mandatory during execution. Since ZebraNet equips wildlife animals that freely move with sensor nodes, the encountered network topology is highly dynamic. Software can therefore often be received in incomplete bundles of packets, rendering simultaneous re-programming of all nodes impossible. The Application Updater serves as a management component for available versions and code bundles.

Although proposed to serve as a general purpose middleware, Impala is very explicit about which event handlers have to be present in the system since these are directly integrated in provided modules. One can derive from the specified functionality that services such as the adaptation of application-level protocols are tightly coupled with information available from the routing subsystem as well as the operating system. Porting Impala to a different system certainly involves more than just a re-adjustment to available resources.

The idea of *Sdlib* [35] is to provide a standard library for operations commonly found in WSNs. Basic usage patterns and protocols are to be extracted from applications and encapsulated into separate components which may then be linked by a variety of different applications if needed. Sdlib has been particularly designed for TinyOS, thus features TinyOS modules written in nesC, allowing applications to invoke functionality with wiring the offered modules. Data collection as well as data dissemination serve as examples for application-independent functionality that has been outsourced to sdlib modules. To benefit from sdlib, each sensor node participating in the network has to be flashed with the sdlib runtime engine. This is a core management entity supplying auxiliary service such as a data flow component, a simple memory management component or a component to guarantee reliable transmission of messages.

A toolbox of services controlled by a runtime subsumes sdlib features. Sticking to nesC code and the TinyOS operating system naturally restricts sdlibs applicability and portability, especially in respect to the RUNES approach. On the other hand, disassembling applications into functional parts to foster software reuse greatly eases application development and debugging, making Sdlib an interesting approach when composing TinyOS applications.

3.7 Dispersed Structuring

Significant support to cope with distribution is the characteristic attribute of approaches classified under the key *Dispersed Structuring*. The ultimate goal is to

offer a purposive additive that enables a lightweight implementation of distributed algorithms. As such, the provided means are not part of the utilized programming language but rather take the form of an API or conceptual model developers can rely on.

The latest addition to the Lime family of middleware platforms *TeenyLime*, proposed by Costa et al. [38], offers a data-centric view upon nodal distribution. Instead of relying on an explicit group or neighborhood primitive as proposed by approaches in Section 3.2, transparent and data-centric interaction among nodes is the key abstraction mechanism utilized. Specifically designed to enable sophisticated sense-and-react applications in wireless sensor networks, it allows for data sharing among neighboring nodes with its tuple space implementation. Stateful coordination with the possibility to reliably share data, the ability to specify multiple tasks and support for reactive interactions are the main benefits that TeenyLime provides to an application programmer.

Central to the TeenyLime implementation is the tuple space, a shared data repository which can be accessed with read and write commands to insert and retract data tuples via pattern matching. The local tuple space of a node is automatically shared with its one-hop neighbors and therefore serves as a communication primitive. For instance, nodes can publish their ability to provide sensor data by putting a special tuple that indicates this ability into the tuple space. When another node wants to invoke a reading, it matches the pattern of this *capability tuple* and will be automatically provided with the data sample requested. Furthermore, the TeenyLime API specifies commands to add and remove reactions whose action parts will be triggered upon the emergence of a tuple. Stateful coordination is also transparently supported by the introduced reliable operations. The TeenyLime middleware is implemented in nesC on top of TinyOS.

Since data sharing in a local context is facilitated by the tuple space interface and due to a simple API that TeenyLime offers, a developer can benefit from relying on this middleware implementation when localized interactions have to be coordinated, as well as reliable networking is necessary. TeenyLime clearly addresses distribution challenges in a sense-and-react context, but is not a general runtime that ensures memory bounds.

Agilla [51] provides an abstraction for wireless sensor networks that relies on mobile agents. A special runtime environment featuring a tuple space for asynchronous communication between multiple agents residing on a host and an execution platform for agents are the core features of Agilla. The general idea is to be able to deploy a vanilla sensor network that only features the Agilla runtime environment. Later on, different applications may be inserted into the network with the help of agents encapsulating application logic. These agents autonomously move around the network to gather data or coordinate local tasks. Agent specification relies on the Maté instruction set, enhanced with specialized instructions to support agent migration, agent cloning and tuple space modifications. Naturally,

the Agilla runtime environment is implemented in TinyOS.

To permit a utilization of agent-based application development in meshed environments, thus couple several sensor networks using an IP-network in between, Hackmann et al [69] combine Agilla and Limone, an agent platform capable to support more elaborate devices than sensor networks, into Agimone. Cross-network interaction is based on intelligent gateways: Each WSN is associated with one dedicated gateway to enable its advertisement to other participating networks. For an Agilla agent to migrate to a distant WSNs, it has to be wrapped into a Limone agent at the gateway, transferred across the IP-network relying on Limone migration and unwrapped and re-injected into the target network.

Using Agilla or Agimone to implement sensor network applications can be useful when multiple applications have to utilize the same network that are not known at deployment time. Since agents provide a mechanism to transfer both code and state across networks, the provision of services encapsulated into distinct agents can be beneficial: Service placement strategies may adopt to current load, energy or resource utilization. On the downside of this approach, communication costs and processing time are much higher for transferring agents than for simple message passing. Applications that basically feature data streaming operations will eventually suffer from these increased cost.

3.8 Conceptual Evaluation of Middleware Approaches

All middleware approaches, along with an intuitive measure to depict and judge their prevailing design rationale, are summarized in Table 3.1. The chosen scale ranges from - , assigned to those middleware proposals that do not address a given dimension at all, up to ++, which denotes a very strong emphasis on the corresponding abstraction. In between these extremes, 0 is used to tag implementations that comprise slight tendencies towards offering supportive measures, whereas + stands for a genuine, conceptual integration of the abstraction mechanism in question.

Once again it is important to point out the conclusions that can be drawn from the table to avoid its misinterpretation. A higher score in a given dimension does *not* automatically make an approach the better choice for any application. Rather, this measure reflects the degree of commitment or dedication to the specific problem area and the resulting level of abstraction. Judged from an application programmers point of view, one can generally conclude that the higher the score, the less previous knowledge about intrinsic challenges is assumed, and the better the approach is suited for rapid prototyping. Clearly, approaches that provide support to more than one end of the abstraction spectrum by means of a stringent conceptual model to interleave domains are definitely superior to both single target or what we call patchwork approaches. An instance of the second group is e.g. the basic implementation of *Logical Neighborhoods*. Recall that it facilitates data sharing among logical neighbors, giving the developer a nice way

Table 3.1: Characteristic abstraction mechanism of middleware approaches

Approach	Support	Distribution	Language
Hood	0	+	-
Logical Neighborhoods	+	++	0
Abstract Regions	-	++	-
TinyDB	0	++	+/++
Regiment	0	++	+
Kairos and Pleiades	+	+	+
Role Assignment	0	++	0/+
Maté	0	0	+
OSM	0	-	+
DESAL	-	0	+
TML	+	-	+
SNACK	++	-	+
Runes	+	-	0
Impala	++	-	-
Sdlib	++	-/0	-
TeenyLime	+	+	0
Agilla and Aginome	+	+	-

Implementation of abstraction by approach:
- = none 0 = low + = medium ++ = high

to think about the structure of a network. When it comes to tasking the nodes though, this point of view cannot be applied any more, thus the model breaks, leaving a developer with programming individual nodes again. A good example for a cohesive application of an adopted model is the *TinyDB* framework: The network is always treated as a distributed database, and therefore no means for individual, inter-node communication is incorporated in the query language. On the downside, this of course limits the influence a developer can have on network setup and communication costs to zero.

As can be derived from this, abstractions are always in the crossfire of being too high, thus preventing application-level influence on cost parameters, or too low when not being able to significantly impact on simplifying application development. Their utilization naturally comes at the risk of hidden costs in terms of performance loss, timing prerequisites or message overhead, thus factors not included in a qualitative evaluation. The quantitative metrics studied for middleware approaches in wireless sensor networks differ not only among abstraction

dimensions, but also target platform, operating system and even implementation, and are therefore not included in the above evaluation.

3.9 Functional Evaluation of Middleware Approaches

The goal of middleware is to provide the application developer with comfortable means to implement applications. Thus, this section discusses the presented middleware approaches in respect to their appropriateness for the development of heterogeneous, distributed sensor network applications. Therefore, we analyze how far common building blocks shared by many application instances are directly supported, which will be briefly introduced in the following. In contrast to the dimensions evaluated in the preceding paragraph, we now switch from a conceptual to a functional discussion of representative middleware approaches.

3.9.1 Common Application Building Blocks

Although the design space for putting applications on top of wireless sensor networks into practice is vast, several usage patterns enjoy frequent employment. These patterns comprise a variety of ways how and where to deal with sampled data, address post-deployment interaction with sensor nodes or the entire network respectively, and explore general concerns of dealing with heterogeneity of deployed devices. In how far middleware approaches explicitly foster one or more of these patterns reveals functional dependencies between classes of applications and the middleware instances, thus allow a functional estimation of appropriateness of a certain middleware for a class of application. To emphasize the relevance of given patterns, well-known deployments and applications associated with it are pointed out.

Data Streaming Pattern (DS)

Streaming of data from multiple nodes to a dedicated sink for the sake of data collection is a common task performed in sensor networks. A read-only request is issued, possibly filtered by means of predicate specification on values, position or participating nodes and processed within the network. Depending on the application, streaming may be scheduled as an automatic, periodic task, e.g. for continuous supervision of data evolution, issued in an ad-hoc or even in an event-centric manner. A precondition for enabling data streaming in a multi-hop environment is the existence of a route from every node to the sink, thus a working routing protocol that data streams can rely on.

Real-world examples following a data streaming paradigm that have been built with wireless sensor networks include predominantly environmental deployments such as [103, 140, 149] or [106]. Here, the network has been used to learn more about the climate, eruptions of volcanoes or the behavior of birds. Especially in this domain, infrastructure is often not directly accessible and phenomena are

usually spread over a certain region, making wireless sensor networks a valuable tool for data acquisition. Other examples that rely on data streaming include structural health monitoring of buildings or bridges [156] or tracking applications that strive for providing fresh information on the geographic position on certain persons [11].

Sense-And-React-Pattern (SAR)

Another usage pattern for wireless sensor networks can be summed up under the keyword *Sense-And-React-Pattern*. Deployed to monitor their spatial vicinity, filter relevant changes and react in a predefined manner to recognized events, sensor nodes can act as remote, distributed guards on physical phenomena. Therefore, data in SAR usually resembles a limited, local scope of validity and triggers local actions denoted by predefined control laws. These control laws can manifest themselves in simple events such as a value passing a certain threshold but can also be complex conjunctions of multiple spatio-temporal conditions. To add robustness to the system, thus avoid triggering a wrong action based on a faulty sensor reading or due to the spatial distribution of an event, values are often collected from multiple nodes physically close to one another. Coordination and control of localized interaction and a meaningful and rich language to express events are major challenges in SARP.

Heating, ventilation and air-conditioning (HVAC) applications [44], event detection and classification [154] or reactive sampling as proposed in [27] are real-world examples that follow a SARP paradigm.

Read-Only-Pattern (RO)

The potential for localized data acquisition is a key reason to utilize sensor networks. For many applications, it is sufficient to have read access upon the sensor node, thus solely gather and process available data. In this case, stateful adaptation of system parameters as well as user interaction after deployment are not integral parts of the application design.

Practical implementations of such read-only applications are for instance the vineyard monitoring experiment [16] where sensor nodes were densely deployed to monitor temperature variations, or the setup for habit monitoring on Great Duck Island [100]. Applications involving passive RFID chips can be seen as the prototype systems that constitute themselves on read-only patterns.

Read-Write-Pattern (RW)

While in a data streaming context status information is predominantly transmitted from a data source to a sink, many applications need an uplink to the sensor network to operate in a useful manner. From a user perspective, this uplink can either be used to tweak parameters such as the sensor sampling frequency or to provide software updates on deployed sensor nodes. Therefore, we categorize

middleware approaches to support this pattern whenever an application is capable to adjust system behavior dynamically during the time of deployment.

Volcano monitoring [149] is e.g. an application that makes use of a read-write pattern by adapting the sampling frequency on event recognition to assure a good trade-off between data granularity on the one side and energy exposure on the other. Another example of read-write pattern necessity is an emergency deployment for road tunnel supervision, proposed in [37], that explicitly enables dynamic loading of new software components onto deployed sensor nodes to reconfigure usage scenes.

Entity-Processing-Pattern (EP)

Node-level processing has to be available on any sensor node participating in an active manner in a sensor network application beyond pure data acquisition. The entity-processing-pattern is in contrast not concerned with whether node-level processing is physically possible, since this is taken as a given fact, but rather states that the application requires selective tasking of individual nodes. The selection can be based on node addresses as well as distinct attributes a node exposes to the application.

Once again, a use case that makes heavy use of the entity-processing-pattern is the (HVAC) example: Water sprinkler activation based on previously detected smoke in a room should be manageable at node-level with specific selection to ensure case-sensitive, precise behavior.

Group-Processing-Pattern (GP)

Unlike Internet-scale networks, the goal of a single node in a wireless sensor network is often of minor importance. Instead, many applications rely on a network scenario which is set up to serve as an entity rather than individual nodes pursuing their own determination, an understanding that calls for new addressing and data manipulation schemes. To achieve coordinated behavior among network subsets, corresponding primitives or services have to be provided transparently to ease application development. Prominent use cases that will benefit from such patterns include retasking a selection of nodes sharing a common attribute, e.g. similar sensors or physical proximity or distributed event recognition.

A running implementation featuring group processing schemes has for example been tested in EnviroTrack [23]. The presence of an object, a distributed environmental event recognized by a group of sensor nodes, is associated with one logical instance maintained by the network, which is in turn used to implement object tracking.

Heterogeneity and Internetworking (Inet)

While the patterns presented so far purely address activities internal to sensor networks, internetworking and heterogeneity become integral issues when real-

izing ubiquitous computing in a mesh-network context: Not only sensor node capabilities may vary among deployed nodes in the network regarding available sensors or actuators, but applications may span over different classes of devices that differ in magnitudes concerning general processing and storage performance. These problems, along with the provision of means to transparently cross network boundaries, are commonly addressed with the help of middleware by hiding the underlying layers and exporting a system-independent interface for service invocation.

Experiments that target heterogeneous hardware include e.g. a road tunnel disaster management application that involves interaction of sensor nodes with handheld and server-sized devices proposed by Costa et al. [36] as well as the e.g. tracking of firemen as studied in the FeuerWhere project [13].

3.9.2 Discussion

Table 3.2 depicts an overview of the discussed middleware approaches and their functional analysis based on the patterns compiled above. Once again, the focus is on typical requirements for wireless sensor network applications and their support by the middleware approaches with their choice being to some extent arbitrary. The provision of patterns is discriminated in two classes: • marks patterns explicitly supported by an approach, whereas ◦ marks patterns which can be possibly implemented with the associated middleware approach, but are not in its focus.

As can be derived from the applications discussed, real-world sensor network scenarios tend to either use the network to acquire spatio-temporal data under harsh conditions, thus in unaccessible regions, over a long period of time or in a high granularity, or deploy the nodes to trigger in-network reactions e.g. via actuators to enable decentralized system behavior. Within this design space, the former practice usually incorporates a means for streaming information, possibly filtered, to a central entity for processing and storage, thus makes use of the data streaming pattern, while the latter requires primtives for event and action semantics, thus relies on the sense and react pattern. Therefore, it is not astonishing that almost all approaches offer implicit support for at least either one of these basic utilization patterns.

Another obvious observation is the fact that approaches designed to overcome distribution issues commonly implement a group processing pattern. A noteworhty exception are those implementations subsumed under the keyword dispersed structuring: Here, support is shifted from a logical/physical networking point of view to the utilization of a data structure which disguises distribution.

An interesting factor for evaluation is whether the middleware explicitly grants write access on sensor nodes. Basically, this parameter has been chosen to reflect the adaptability of approaches at deployment time, so to what extent an interface is available for applications and/or users to dynamically adjust system behavior based upon network state. It is clear that the lack of such write access mechansims

Table 3.2: Functional analysis of middleware approaches.

Approach	DS	SAR	RO	RW	Inet	EP	GP
Hood		○	●			○	●
Logical Neighborhoods	●		○				●
Abstract Regions	○	○	○				●
TinyDB	●		●	●		○	●
Regiment	●		●				●
Kairos and Pleiades			○			●	●
Mate	○		●	●		●	
OSM			●	●		●	
DESAL	○	●	○	○		○	
TML		○	○	○		○	
SNACK	●		●	○		○	
Runes	●	○	○		●	○	○
Impala	○		○	○	○	●	
Sdlib	●		○			●	
TeenyLime	○	●	●	●	○	●	
Agilla and Aginome		○	○	○	●	●	

● Explicit support
○ Implicit support

DS = Data Streaming
SAR = Sense-And-React
RO = Read-only
RW = Read-write
Inet = Internetworking
EP = Entity Processing
GP = Group Processing

at the middleware layer does not necessarily doom applications to static behavior: if the middleware is e.g. accessible via a dedicated API, the application can simply bypass it and invoke underlying system functionality to achieve the requested behavior. Explicit support however is rather scarce and cannot be attributed to a single dimension of abstraction. A slight tendency towards approaches implementing a language abstraction can be observed, but by no means can be categorized to be of prime interest in this abstraction dimension.

Finally, key assignments such as addressing heterogenity of devices and provision of end-to-end service invocation even accross networking domains cannot be confirmed to be the driving forces for middleware concepts in wireless sensor networks: only a quarter of the investigated approaches address these challenges, with only half of them considering them to be issues of prime importance.

3.10 Critical Evaluation

A lot of effort has been put into making wireless sensor networks accessible for a broad community beyond experienced programmers and sensor networking experts. While the last two section thoroughly discussed general motivation and in-detail conceptual and functional parameters for each approach, this section is taking a step back, evaluating the overall merit of proposed abstraction categories with respect to a programmers' expectation. Doubtlessly, these expectations may differ dependent on the actual expertise of a developer and the specific application or class of applications one is eager to build, however the following, very basic demands are unlikely to change.

- *Reliable and robust software development*: First and foremost, software has to work. Reliable and robust software is the fundamental key for successful deployments. As a consequence, a valuable abstraction shields a programmer especially from those errors that arise from intrinsic difficulties of the target domain that go beyond his usual experience background as these are hard to track.

- *Fast prototyping*: Independent of the actual project, cutting development time is a concern which directly influences overall project costs. Naturally, the earlier within the development process design decisions can be tested and explored, the better can the software be targeted to individual requirements. Support for rapid prototyping is thus an important feature that middleware platforms should address appropriately.

- *Dedicated support throughout the software development process*: The availability of a nice, conceptual abstraction is great, but practically irrelevant if not embedded within a reasonable tool chain which allows for its exploitation. Similar to the demand for fast prototyping capabilities, the software engineering perspective should not be disregarded.

Looking at the distinct categories of middleware approaches, the central question which challenge is *the* foremost concern that hinders reliable and robust software development yields two answers: Either, approaches emphasize that it is the distribution and networking issue that developers face, or they target to overcome the concern of software development for embedded hardware.

To our understanding of the problem domain, the second problem is the most critical challenge to address for reasons of ease of development and mere software size. Although abstractions that offer mechanisms for improved development of distributed applications substantially reduce overall programming effort, this is only the case when applications can directly benefit from the offered mechanism. Naturally, the provision of a build-in routing protocol suits e.g. solely those applications that not only rely on a data streaming pattern, but also feature the network topology, thus associated roles of individual nodes, the approach foresees. Similar

conclusions can be drawn for e.g. the prominent neighborhood abstractions: Localized interaction is certainly an important building block of many applications, but once again not a key concern shared by all. Once the functionality supported is *not* a central concern of the envisioned application scenarios, or only to a small extend, it is questionable whether the constraint memory resources should be occupied by the associated middleware abstraction, or whether the deployment of an application-specific implementation is not favorable.

In contrast to this, embedded hardware with its emerging challenges are a fundamental reality shared by *all* sensor network applications. One has to be aware that this is not to say that distribution and networking are note equally important issues, yet the nature of *how exactly* this manifests itself is a lot more specific to individual applications. Therefore, objecting the intrinsic problems that arise from embedded sensor node tasking, that hamper software reliability and application robustness in the first place, is the more appropriate choice.

Returning to the initial abstraction dimensions and planes that approaches can be classified within, see Figure 3.1, this request is predominantly objected by those that foster improved capabilities for concept mapping, as well as those that in addition provide supportive measures. A closer look at the proposed abstractions in these domains reveals however that none of the suggested solutions is able to fulfill all basic demands denoted above, each lacking at least one of them. The least valuable abstraction is certainly DESAL, since, to the best of our knowledge, no implementation exists at all. Maté, even though providing a virtual machine to shield a programmer especially from the rather demanding TinyOS event model, exposes only a very limited set of low-level instructions to a developer. This hinders rapid prototyping in case more than standard sense and send operations are needed. Likewise, relying on TML as a basis for sensor network application development is only partially satisfactory. It demands for application-specific adaptation by design, a circumstance however coherent as it targets to serve as an intermediate language.

Among all proposed approaches, SNACK is the one that objects both the request for rapid prototyping and for reasonable support in terms of software engineering capabilities. Simulation and evaluation tools developed for TinyOS can directly be utilized since SNACK is a framework for TinyOS service composition. On the downside, the interface to programming is still the nesC language with the underlying TinyOS operating system, leaving low-level concerns such as stack management and event-centric task decomposition up to the programmer. These in turn are explicitly addressed with the OSM programming model: Managed state variables and stack management definitely facilitate application development and at the same time ensure software robustness. But then again, only a prototypical implementation of an OSM compiler, and no support for pre-deployment testing and evaluation is available.

The gap of missing a programming framework which addresses embedded, event-centric programming challenges, but at the same time provides substantial support throughout the software development process still remains to be closed.

3.11 Concluding Remarks

The goal of this chapter has been to provide insights on the current state of the art in middleware developments targetting wireless sensor networks. Unlike well-known middleware approaches utilized in classical distributed systems, approaches designed for this domain face challenges beyond distribution and/or mobility: the embedded nature of devices that restricts excessive resource exploitation, the unreliable wireless medium as well as the predominantly event-driven operational scheme call for suitable abstractions to enable rapid application development.

In order to offer a thorough evaluation of presented middleware approaches, we depend on a twofold qualitative analysis. First of all, we motivate the chosen selection by means of classifying the perspective on abstraction each scrutinized approach conveys. This way, its inherent conceptual point of view becomes apparent. The presented classification is complemented by a functional analysis to precisely elaborate the actual benefit an application programmer can expect to experience when building upon a chosen platform. To this end, we base our evaluation on common usage patterns shared by and derived from real-world sensor network deployments. With both metrics at hand, future developments can quickly be evaluated and easily be put into the hitherto existing context.

The analysis has also revealed that those approaches that object distribution problems have reached a very mature state as arising problems are addressed in a multitude of different flavors. This can however not be attested to those middleware implementations targeting to offer dedicated abstraction from sensor node tasking. Especially in case supportive measures throughout the development process are needed, a major gap still to be closed becomes apparent.

Chapter 4

The FACTS Middleware Framework

The last chapter presented general concepts and contributions to grant assistance in tedious sensor network programming, followed by a review of state-of-the art approaches to overcome WSN specific challenges. As has been pointed out, an approach that comprises both a strong conceptual model to support robust software development and a holistic framework has up to date not been available. In this chapter, we introduce our solution to object this gap, a middleware framework called FACTS. The core idea for designing FACTS has been to provide a simple but versatile model for dealing with both an event-based execution environment and wireless communication using a unified data model. A developer is empowered to describe node-local behavior by means of rule specification. Composed of a condition part stating under what circumstances or *event* a sensor node is requested to become active, and a corresponding action part automatically executed upon condition verification, a rule becomes the basic programming primitve to prompt individual nodes. The motivation for depending on this declarative programming paradigm has been the straight-forward mental model the choice offers to cope with the prevalent event-centric usage pattern of sensor networks: Rules provide an intuitive, modular and simple handle to specify reactive programs, thus have the potential to allow for fast and flexible implementation of application concerns.

To alleviate programming efforts, the FACTS framework comprises a comprehensive programming model, a suite of programming tools for rule compilation and debugging and a tailored runtime environment for the sensor nodes to shield the programmer from intrinsic hardware, data and communication matters. The remainder of this chapter will briefly give an overview of the FACTS framework and associated supportive tools in Section 4.1 and present basic language primitives in Section 4.2 to highlight its general idea. Section 4.3 applies the insights gained to point out how FACTS can be conceived with regard to the previously evaluated approaches. In-detail information on addressed framework components concerning design, implementation and performance will be provided in subsequent chapters.

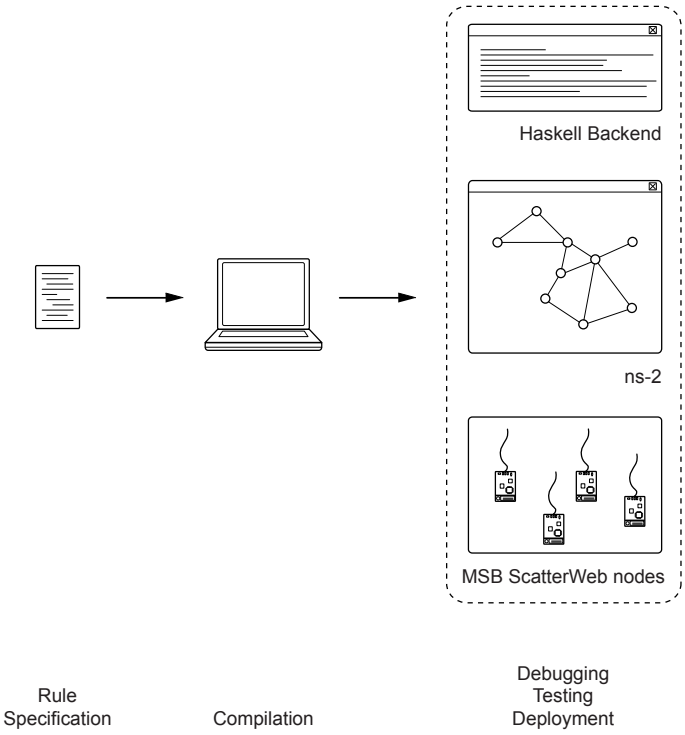


Figure 4.1: Facts framework toolchain

4.1 FACTS concept and components

Driven by the quest to enable rapid sensor network programming with a clear abstraction from both event-centricity and embeddedness, FACTS offers a rule-based programming language that can be interpreted on various platforms. Figure 4.1 gives an overview of the usual steps involved in application development as well as the core elements available during this process.

4.1.1 Application development with FACTS

At the beginning, the envisioned behavior of sensor nodes is described with sets of rules specifying what events are of interest and how to react upon their detection. Events may be simple such as the sudden availability of a new sensor reading or the reception of a packet over the radio interface, but can also comprise complex time, state and data dependencies. Events, stateful information and user-defined data, thus any data visible to an application programmer, is wrapped into a so called *fact*, a mere list of attribute/value pairs tagged with a name and additional system-generated information. The *ruleset(s)* that outline rule/fact interaction are then input to a ruleset compiler, see Section 6.1, which in turn ouputs a concise

bytecode dependent on the target platform.

FACTS supports three different backends. First of all, a `Haskell` [138] backend with `Hugs` [5] being the corresponding interface can be used, which enables debugging of developed rulesets at a very fine granularity and allows for formal reasoning about the language. A developer can specify a network of nodes runnning the same ruleset and a flow of events that enter the system at a certain time, then step through the bytecode and trace the flow of execution. Further compilation targets include the ScatterWeb sensor network platform and the `ns-2` network simulator [49]. In both cases the compiler produces basically the same output solely differing in their individual representations. This bytecode can be deployed and afterwards interpreted by the FACTS runtime environment.

4.1.2 FACTS runtime environment

As can be derived from the above, FACTS programs are not compiled into native code, but into a dedicated bytecode which has been optimized for size and deployment flexibility. The architectural model therefore implemented is that of an interpreter or process virtual machine, providing a sandboxed runtime environment for rules. Besides the operating system, FACTS remains the only active process executed on the nodes, with all application-level logic purely supplied in rulesets. The advantages of utilizing such an approach, especially in the context of wireless sensor networks, are at hand:

- *Managed access to OS functionality*: Providing a thin layer of software in between application and operating system allows to guard and eventually to prohibit access to underlying system resources. If crafted well, the interface of a virtual machine will enable a developer to abstract from low-level concerns and concentrate on the application logic instead. Implementationwise, the necessity for manual stack management in embedded programming is in particular one of the big barriers preventing rapid prototyping and stable execution. Shifting this concern to a runtime environment can significantly leverage application design.

- *Supervised scheduling of application-level source code*: The event-centric application semantic predominantly applied within the sensor network domain has been frequently named to be a major challenge in programming [93, 88]. Naturally, asynchronous behavior due to incoming interrupts and a non-linear flow of program execution require programming expertise to achieve the envisioned network behavior. Once again, a virtual machine can export a higher level of abstraction, thus a simpler execution model to the user and shield any scheduling and timing demands from her.

- *Platform independence of application-level source code*: A well-known and often exploited advantage of utilizing interpreted bytecode is its independence of a specific hardware platform. Heterogeneous sensor networks that

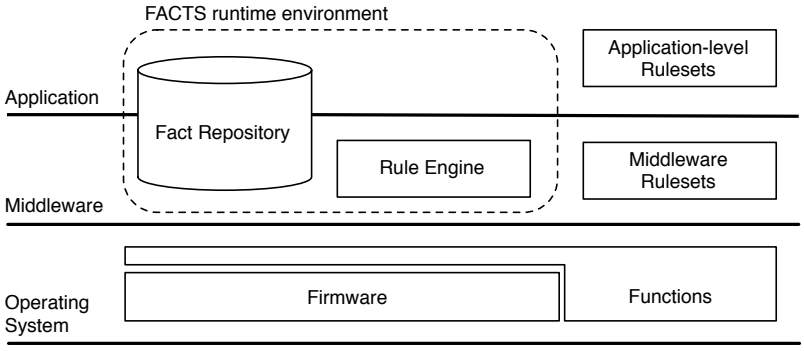


Figure 4.2: Overview of the FACTS architecture

integrate devices of variable magnitude will strongly benefit from a virtual machine solution; available resources can be allocated individually while at the same time a unified programming model facilitates a global view on the problem domain.

- *Increased flexibility for runtime retasking*: Due to the embedded nature of sensor nodes, programming efforts often favor a monolithic, tightly-coupled implementation over a modular one. As a consequence, a clear separation of concerns, e.g. of application-level software and operating system functionality is seldom enforced and therefore frequently bypassed. While this is not recommended from a software engineering point of view, it becomes a real problem when in-situ software updates have to be made available at runtime. Over-the-air (OTA) reprogramming can either be necessary for reasons of scalability or of limited physical access to sensor nodes [85], and involves both software distribution and linking. The tighter coupled systems are, the less intuitive and manageble differential updates of native code become, leading to the distribution of complete images in the worst case, see e.g. Deluge [82] or TinyModules [141] for an in-depth discussion. Also, bytecode can be significantly smaller, making OTA flashing less expensive in terms of energy spent for mere software distribution.

The general architecture of the FACTS runtime environment and its relationship to underlying software components is visualized in Figure 4.2. Each node exhibits a central entity for scheduling rule evaluation, the `rule engine`, and global storage for data, the `fact repository`. Hence, control flow and manual memory management are the intgeral parts of programming whose pecularities are hidden from a developer. The supplied ruleset may either be used to implement middleware functionality such as e.g. routing algorithms or management of neighborhoods or application-level concerns. Note however that this distinction between middleware and application-level rulesets is rather a semantically motivated than an

architectual differentiation. Upon event notification, the rule engine schedules the rules for interpretation: Conditions of rules are evaluated by means of matching against the facts in the repository which will eventually lead to triggering the execution of their action part. This forward chaining approach allows to generate new knowledge in terms of facts, thus shares production rule semantics.

The interface to the firmware is kept simple: Bottom-up, FACTS intercepts firmware callbacks from sensors if requested to do so, user-defined timers and dedicated FACTS radio packets, with return values being pushed as facts into the fact repository. Since interaction with the operating system is also necessary in a top-down direction, e.g. to log data to secondary storage or to control sensor calibration, a thin layer of functions has been introduced that may be invoked from the action-part of a rule.

4.2 RDL language

As pointed out in [66], an important key to good language design is to not forget that languages are designed first and foremost to be used by people, and then interpreted by machines. The language itself determines how a problem will be conceived, and in response, how easy it is to be described in the corresponding language. Directly related to this quest for usability is the aim for language brevity: Flexible, basic programming constructs are preferable when yielding fast prototyping and solid, maintainable sourcecode. While a broad language API with various data structures directly integrated into the language itself can be beneficial for general-purpose programming languages, it is certainly neither mandatory nor necessarily better for domain-specific language design such as targeted in this context. Naturally, the chosen data and programming model have a great impact on language expressiveness - as experienced programmers usually strive to have as much control as they can to impact their implementation according to their needs, the main challenge for language design is to balance abstraction, brevity *and* low-level control according to the envisioned operational purpose.

4.2.1 Basic Building Blocks of RDL

Taking the challenges that software development for wireless sensor networks bear from the introduction in Chapter 2 and relating them to the general demands presented above, the objectives for designing the ruleset definition langauge (RDL) were clear: The programming abstraction has to be
(1) small and concise,
(2) reflect the event-centric model employed to operate these networks,
(3) feature a very general data model suitable to unify computation and communication, and
(4) nevertheless provide a powerful interface for device-level interaction.

In the following, only a small overview of RDL concepts is provided, closing with a concrete example of RDL rule syntax. Chapter 5 features a detailed review of the language, with the corresponding grammar being additionally provided in Apppendix A and exemplary implementations in Appendix B. Details on the actual implementation of rule evaluation strategies and processing concerns will be the topic of Chapter 6.

Rules

To implement the event-centric processing scheme, thus solely react to relevant changes of the environment, rules are a perfect match [19]. They nicely mirror the push semantics prevalent in wireless sensor network applications, where incoming data obtained from sensors or received via the radio interface triggers reactions on a node, whereas otherwise nodes turn to low-power mode for power efficiency. Reactions may subsume filtering actions, aggregation schemes, chains of data processing steps which distill high-level or complex events from several low-level data items or simple reactions that trigger actors or start new routines on a node. As a consequence, a rule is the basic computational entity available in RDL.

RDL rules are reactive rules, sets of conditional actions with conditions being indicated by a left arrow (←) and statements, marked by a right arrow (→).

```
rule name priority
<- condition [condition_list]
-> action [action_list]
```

RDL rules are neither classical production rules, although their *WHEN condition THEN action* structure may convey this assumption, nor pure ECA rules but integrate characteristics of both worlds into one approach. Reactive rules generally correspond to change. While production rules trigger upon a modification of their working memory, thus actively maintain rule engine state, ECA rules solely depend on rule-local state, defined by objects and events matched by a rule. Therefore, the semantics of reactivity and its implementation are slightly different: ECA rules, typically denoted in an *ON event IF condition THEN action* format, explicitly name and model events in contrast to a state-driven approach pursued by production rules.

RDL rules are best described as production rules implementing an explicit event concept, a circumstance that can be attributed to its data model. Both events and state are fused into the single data abstraction of a fact, see also the following section, but can be discriminated via information on their state tagged to each individual fact. The sum of all rules that form a semantical entity is referred to as a `ruleset` in RDL. To ensure confluence and observable determinism of rules a priority has to be applied at the time of definition [6]. Priorities do not have to be unique, however non-deterministic scheduling of rules tagged with the same priority is then accepted.

Facts

An expressive, yet simple and versatile data model is an important basis for successful language design. Similar to the database community, we chose to use a declarative, tuple-based data abstraction which furthermore yields to unify the representation of computation and communication. The idea has been to render the origin of incoming data obsolete, unless explicitly requested by the envisioned application. In general, the reaction to a specific data item has to be the same, whether it has been obtained from sensors on the node itself, it is a result of a successful production run of deployed rules or whether it has been sent by a neighboring sensor node. Due to their versatility, we chose so called *facts*, named tuples of typed key-value pairs (called *properties*) to serve as the basic data model. For instance, a fact representing a reading of a temperature sensor on a node may be expressed as

```
temperature [value = 20]
```

Facts are not only a straight-forward way to specify application level data, but also allow for wrapping low-level data such as sensor readings, timer interrupts or return values of library calls into a distinct format. Pattern matching used during rule condition evaluation can simply be applied on the facts themselves as well as on their corresponding properties. Note that this data and processing model emphasizes *content-based* data handling rather than demanding for an instruction-oriented processing scheme.

Each fact can be understood as a global variable with global scope, thus is visible throughout the deployed ruleset. Note that the difference to global variables utilized in imperative programming is however crucial: Memory can be *conceptually* de-allocated at runtime simply by deleting a fact from the repository, and reused for allocating other facts by subsequent rules. Within traditional event-processing systems, efficient memory utilization is a genuine problem as actions often times have to be implemented in a split-phase manner, see also 2.1.3. Due to stack unrolling, automatic variables cannot be used to share program state leading to a massive declaration of global variables that however tie up valuable resources over the complete execution time for a program.

In contrast to other rule languages and motivated by the target domain being resource-constrained devices, RDL does not support any rule-local, automatic variables. While this naturally causes some inconvenience since operations to bind and thus re-use results of pattern matching operations to local variables are not available, the benefits however outweigh the drawbacks. Given solely global data storage available for a developer, the stack size, a major cause of runtime errors and system crashes during sensor network deployments, can be completely supervised by a runtime system. Typically, a certain amount of memory is allocated at compile-time for storing facts, which in our implementation corresponds to the fact repository.

Facts are automatically tagged with their creation or update time, their originator and state, thus whether they can be conceived as events or not, see also

Section 5.2. This way, spatial as well as temporal reasoning is possible, and high-level, complex events may be specified within rule conditions.

4.2.2 Conditions: Guards upon Reactivity

Within the condition part of a rule, a developer can denote the circumstances that lead to rule triggering. In detail, supported types of conditions include test for existence of a fact of a specific type (via the declaration of an `exists` condition), evaluation of fact properties (via the definition of `eval` conditions) and conditions that specify unary operations on the set of all available facts (e.g. `sum` and `count`) or on dedicated fact properties (e.g. `min` and `max`). Pattern matching of available facts to given constraints will then be applied to resolve the conditions and eventually return a boolean value indicating the result of this evaluation process.

The quite limited set of operators available for condition specification is however very powerful as soon as operators are nested or combined sequentially. Threshold specification for certain fact properties is available as well as range queries applied to a filtered subset of facts or the definition of conditions for temporal or spatial relationships between individual facts. With these at hand, sophisticated condition specification is supported, or more precisely adequate filtering mechanisms on the set of available facts are available.

4.2.3 Statements: Combining general and domain-specific demands

The action part of a rule, composed of one or more statements, serves two different requirements: the definition of common data manipulation instructions, but also the specification of what can be subsumed under the keyword of *domain-specific instructions*. The former simply resemble well-established schemes from database systems, where insertion, update and deletion of rows, columns or single tuples in tables can be applied. In general, their counterpart in RDL, the `Define, Set` and `Retract` statement, operate on the set of facts that are filtered via their name, which in turn yields set-based manipulation granularity. However, with the application of filtering conditions, manipulation granularity can of course be broken down to individual facts where required. With these statements at hand, a programmer is hence able to denote new facts needed to encapsulate program knowledge, modify individual properties of available facts and erase any data items that have become obsolete. Once again, we'd like to point out that no local variables exist, and therefore any assignment of new values to properties or fact filtering inquiry depends on pattern matching mechanisms. However, it is possible to name specific filtering patterns to foster code readability which are then referred to as `slots`.

The more interesting set of statements is directly related to the actual interface a wireless sensor node exposes to a developer. Typical hardware components that

are available and need to be integrated into a domain-specific language abstraction include a variety of sensors, usually chosen according to application needs, a wireless transceiver and some kind of interface to a hardware timer. Optionally, sensor nodes may be equipped with additional, secondary storage such as an SD card, actuators mounted on the node itself to allow for ad-hoc physical reactions or modules for audio-visual feedback (e.g. a beeper or LEDs).

When embedding access to these components into the language it is necessary to allow for as much low-level parameter tweaking as possible to not prohibit application fine-tuning, but at the same time grant usability within the frame of the language. Since configuration concerns for the wireless transceiver are typically the same regardless of the actual transceiver model[1], communication requests can simply be expressed by issuing a `Send` statement. Here, a developer has to clarify where to send a certain fact, so either in a unicast or a broadcast manner, and whether the transmission power is to be explicitly set. Note that the chosen data model allows to abstract from actual packet transmission since the burden of wrapping or segmenting data is shifted to the responsibility of a runtime environment.

However, all other hardware components may substantially differ in terms of available parameter space for an actual unit, even in case they belong to the same device category. To cope with this problem, any other low-level functionality exposed by a sensor node is accessible via a `Call` statement. The number of parameters passed when issuing a Call statement can then match the number of possible configuration parameters of the given unit, which will be identified by a keyword made available as part of the language itself. As a matter of fact, this approach requests the adaptation of RDL to a given sensor node platform, as each low-level functionality has to be exported to be callable from RDL. Nevertheless, it enables a very targeted language design which on the one hand allows for extensibility if needed, but still respects the limits that embedded devices impose on software development. A detailed overview of the interface design will be given in Chapter 6.3. As has been pointed out before, return values of statements (e.g. a sensor sample previously requested) are wrapped as facts upon completion of the corresponding function call, asynchronously pushed into the runtime environment and made available for subsequent rule evaluation.

A last category of statements available in RDL are those that manipulate fact state, represented by the `Flush` and the `Touch` statement. In case explicit flow control has to be integrated into a set of interacting rules, these statements provide the needed interface, see also Section 5.3 for more details.

4.2.4 Rules in Action

The rather abstract presentation of language constructs denoted above is able to convey the basic idea of programming wireless sensor nodes with RDL, but certainly lacks a way to present the actual elegance of using the language. Consider

[1] We assume the utilization of RDL for the implementation of protocols situated *above* the MAC layer, and export simple MAC layer characteristics to the programmer.

Listing 4.1: The `Hello World` program as an RDL rule.

```
1 ruleset HelloWorld
2 /* Incoming hello facts could have the following structure
3  * fact "hello" [val = "world"]
4  */
5
6 rule printHello 100
7 <- exists {"hello"}
8 -> call printFact({"hello"})
```

therefore the RDL implementation of the classical HelloWorld program presented in Listing 4.1.

The ruleset `HelloWorld` contains a single rule called `printHello` which is labeled to have a scheduling priority of 100 (a number that is used to determine the evaluation order of rules upon the occurrence of a new event, see Section 5.2 for details). Whenever a `hello` event is recognized, thus a (new) fact named "hello" appeared, the rule triggers and calls a function named `printFact` from the set of exported system functions. This in turn is instructed to print all facts and their corresponding properties that match the name `hello` to standard output.

Several interesting observations can already be derived from this small example: First of all, the way the rule is denoted, it is completely oblivious to the origin of the `hello` fact. It may have been received over the radio interface from a neighboring node or issued by a firmware function due to the execution of a callback. In case this rule is part of a different ruleset that comprises further rules, it will also fire when one of these rules alters a `hello` fact, e.g. by setting the property `val` to "folks" instead. Unlike usual ECA semantics, the event of the occurrence of `hello` is not transient but its existence is preserved in the fact repository (unless explicitly retracted by a rule), a circumstance that allows for a straight-forward implementation of predicates on event sequences.

With the absence of local variables, RDL rule evaluation depends solely and at any time on pattern matching. As a consequence, the `hello` event, which usually concerns only a single fact, will, the way it is denoted in the statement in line 8, result in printing *all* available `hello` facts within the fact repository. There is no implicit binding of the event to some internal variable and passing it between condition and statement part of rules. Rather, every condition and every statement is independently evaluated in regard to the current state of the fact repository.

Ruleset evaluation is always triggered upon the occurrence of a new event which manifests itself in a new or altered fact. Then, all premises of the rules are checked whether they incorporate a reaction to this event and whether all other conditions specified are met to potentially execute their corresponding statements. Unless triggered rules specify productions, see also Section 6.2, the rule engine will return to an idle state until the next event occurs.

Listing 4.1 serves to provide only a first impression on the language itself. Detailed information with more sophisticated examples will be made available in subsequent chapters of this thesis.

4.3 Qualitative Evaluation of the FACTS middleware framework

To sum up the brief introduction of the FACTS middleware framework presented above, it is interesting to compare and relate its features to the approaches that have been discussed in the preceding chapter. Recall that mechanisms to achieve abstraction for tasking wireless sensor networks have been roughly categorized into the three dimensions *Support*, *Distribution* and *Concept Mapping*, each of which addressing a key challenge in WSN programming.

The FACTS middleware framework incorporates with RDL a domain-specific language for WSNs (thus alleviates the actual process of mapping problems to sourcecode with the help of a language), and a versatile runtime environment (thus offers substantial support to a software developer). Languagewise, rules and rulesets are per se modular programming constructs that, in addition to reflecting the event-driven aspect of programming, allow for a very fine granularity of code composition. Due to its focus on improving the node-level programming interface exposed to a programmer, FACTS does *not* provide any explicit mechanism to overcome the challenging pecularities of distribution. This design choice can be attributed to the fact that it has not been possible to derive one dedicated interaction scheme valid for all kinds of wireless sensor network applications or protocols. Often times, the implementation of e.g. transparent routing or neighborhood management schemes are simply not needed, but automatically integrated into approaches and thus flashed onto the sensor nodes. Naturally, precious memory is then consumed for non-functional parts, a circumstance that to our understanding has to be prevented by design. Therefore, FACTS favors to push distribution issues into dedicated rulesets that may be linked when needed, leaving protocol engineering to the developer.

Hence, it can be classified to belong to the approaches subsumed under the keyword of *Composite Programming*.

Table 4.1 relates FACTS to the other approaches that belong to the Composite Programming category, namely the Token Machine Language (TML) and the Sensor network application kit (SNACK). As has been pointed out in detail in Section 3.5, the former is designed to serve as an intermediate language, offers tokens and token handlers as its basic programming abstraction, but still requires the implementation of application-specific token handlers *per application*. Therefore, rather than emphasizing the language perspective by providing a self-contained language, the focus is on utilizing a distinct mental model in combination with a dedicated execution environment. Language and support perspective can be judged to be equally met at a medium level.

Table 4.1: FACTS in relation to other Composite Programming approaches

Composite Programming	Support	Distribution	Language
FACTS	++	-	++
TML	+	-	+
SNACK	++	-	+

Implementation of abstraction by approach:
- = none 0 = low + = medium ++ = high

SNACK follows a different approach: It is basically a meta-language built on top of nesC to enable better service definition and composition. The SNACK library provides a powerful, component-based API, with the language being a vehicle to combine supplied services. This all yields an implementation of supportive measures by means of introducing an additional layer of software in between system and application, composable via language constructs. Hence, support can be definitely rated to be high with the language perspective being solely of medium importance.

In contrast to these two approaches, RDL is an integral part of FACTS, a stand-alone language that neither forces the programmer to be aware of and understand the underlying target language, which is mandatory for programming with SNACK, nor to enhance the language with specific handlers for each application. The design of the statement part of a rule is a compromise between the quest for language conciseness and the availability of a rich API in native code for means of runtime efficiency. In special cases, e.g. when complex mathematical functionality is required for a given concern, the current API of available, native function calls may not suffice, thus additions to the language may be required. However, for the majority of typical WSN applications, the implemented API will most certainly meet their needs, a fact that substantially differentiates them from TML token handlers.

Programming support is provided at different levels of the FACTS framework. First of all, it is granted at the language level, offering powerful abstractions to express event-centricity and to specify modular software components. In addition, the programmer is also shielded from runtime concerns, including error-prone, manual stack management and the enforcement of the correct event ordering. Due to this, we can attest FACTS to score high not only on the language, but also on the support scale.

Turning to the functional analysis of FACTS in respect to TML and SNACK, no surprises are revealed, see Table 4.2. Key design goals of FACTS have been to enable an improved specification of reactivity as well as a sufficient, yet not restrictive integration of device-level interaction into a high-level programming ab-

Table 4.2: Functional analysis of Composite Programming approaches.

Approach	DS	SAR	RO	RW	Inet	EP	GP
FACTS	○	●	●	●	○	●	
TML		○	○	○		○	
SNACK	●		●	○		○	

● Explicit support
○ Implicit support

DS = Data Streaming
SAR = Sense-And-React
RO = Read-only
RW = Read-write
Inet = Internetworking
EP = Entity Processing
GP = Group Processing

straction. Hence, FACTS can be attested to explicitly support the sense-and-react pattern, the read-write-pattern and the entity processing scheme which reflects its focus on node-level tasking.

As an intermediate language, TML does not export a rich set of features itself, but rather serves as an additional layer of software to abstract from the underlying TinyOS operating system. Since the token handler concept can be utilized to implement reactive behavior in case a token corresponds to an event, at least implicit support can be appointed here. Implicit support can also be attributed to the implementation of the read-only and the read-write-pattern, since they require bypassing the TML abstraction, a statement that equally applies to read-write support in SNACK.

The SNACK library offers several particular implementations to stream obtained data through a network, e.g. a simple, duplicate-surpressing flooding protocol and a tree-based routing protocol with the root being a data sink, indicating direct support. Regarding FACTS, one can argue whether it is correct to assign implicit data-streaming support to it: On the one hand, there is no such thing as a dedicated ruleset for this task that is automatically supplied with the FACTS runtime environment upon flashing it onto a node. Nevertheless, amongst others, rulesets that implement data streaming algorithms are available, see Chapter 7 and may be linked to newly developed applications on demand. Also, addressing device heterogeneity with FACTS is feasible and in fact easier to achieve than for the other two approaches: While their implementations are tightly coupled to the TinyOS operating system, the system interface specified for FACTS is small, and porting the runtime environment only a matter of adjusting this interface and the allocated memory available for fact storage.

4.4 Concluding Remarks

The intention of this chapter has been to put individual building blocks of the FACTS middleware framework into a broad context and provide a brief, introductory overview of all of its components. The core of the framework evolves around the rule-based programming language RDL which offers reactive production rules as a means to express sensor network algorithms. Rule evaluation can be triggered by incoming events, data items that are wrapped into the dedicated format of a so called fact, with the rule evaluation process depending on pattern matching.

Due to numerous advantages that runtime environments are able to offer, and that in our opinion legitimate eventual loss in execution latency, rules are compiled to bytecode and interpreted on a process virtual machine. This runtime environment comprises a rule engine in charge of rule evaluation and scheduling, and a fact repository as the central means of data storage for application-level data. Furthermore, a set of complementary tools that enhance the debugging and testing cycle of software development are part of the FACTS family.

The chapter is finalized by a quick review of FACTS feature in respect to previously introduces programming abstractions. Overall, this qualitative evaluation revealed FACTS to be a decent abstraction in terms of conceptual and functional features, exporting a solid, node-level programming abstraction to wireless sensor network programmers.

Chapter 5

RDL: Rules to rule wireless sensor networks

RDL, the ruleset definition language, has been proposed to facilitate instructing wireless sensor nodes. Similar to any other programming language, syntax, semantics and pragmatics characterize its subtle nature and therefore serve as a key to understanding the language's practicality.

A presentation of its syntax manifests a language's symbols and denotes how they can be combined to specify well-formed words, phrases and sentences thus programs. Hence, the syntax definition reveals the internal structure of a language. Since only programs that are syntactically correct do have a semantics, syntax specification is a natural starting point of language design and evaluation.

Denoting a language's semantics in turn corresponds to giving an explanation of its meaning. Therefore, a denotational semantics specification provides insights on how a proper sentence of the language is supposed to be evaluated by the machine that is tasked to execute a given program. Independent of any issues concerning compilers, machines or interpreters, the semantics of a language are to be universally applicable. During the design phase of a language or when porting it to run on different system architectures, a denotational semantics can be utilized to achieve envisioned language functionality and perserve its execution behavior across platforms.

Finally, language pragmatics reflect the usability of a language. Pragmatics are neither subject to specification nor measurable in a straight-forward quantitative manner. Rather, they mirror the ease of use of a language, its application area and whether stated goals that lead to language specification were successfully met. Therefore, pragmatics can be conceived as the inner beauty of a language, a fact that can be described and pinpointed with showcase applications; evaluation beyond this is subject to empirical, long-term studies.

This chapter is dedicated to discuss RDL language syntax, semantics and pragmatics to provide a detailed overview of how the language works. Therefore, the core language syntax is presented in Section 5.1, followed by a formal denotation

of its semantics in Section 5.2. A discussion of language pragmatics is provided in Section 5.3, where a few design patterns are presented that have been proven useful when utilizing RDL as a holistic programming language for wireless sensor networks. Since the discussion of language paragmatics can nevertheless also be conceived as stating its practical relevance, we will return to this in Chapters 7 in more detail. At the end of this chapter, we will point out relevant steps undertaken to enhance the core language in Section 5.4, and evaluate in how far the introduced additions fill in a gap of missing features in a plausible manner. Section 5.5 summarizes the findings in regard to language design and concludes this chapter.

5.1 RDL core language syntax

Programming languages conform to a context-free grammar. A set of productions maps so called *nonterminals* - abstract symbols from a specific alphabet - on the left-hand side of a production rule, to a sequence of nonterminal and *terminal* symbols on the right hand side. A language therefore corresponds to the set of possible sequences one can generate from repeated application of right-hand side productions to nonterminal symbols, starting from a predefined distinguished nonterminal, the *goal symbol*.

Successive subsections cover a piecemeal discussion of the lexical 5.1.1 and syntactic 5.1.2 grammar of RDL to allow for a comprehensible presentation. A complete specification conforming to EBNF style conventions for reference purposes can be found in Appendix A.

5.1.1 The Lexical Grammar

Denoting the lexical grammar clarifies how input elements from the ASCII character set are combined to form *whitespaces*, *comments* and *tokens* in RDL. Only the latter contribute to the syntactical grammar of the language, thus whitespaces and comments have to be discarded during the translation step of RDL programs from a stream of input elements to a sequence of terminal symbols.

```
input          ::=  [inputelement+]
inputelement   ::=  whitespace
                 |  comment
                 |  token
token          ::=  identifier
                 |  keyword
                 |  literal
                 |  seperator
                 |  operator
```

Whitespace

ASCII space, horizontal tab, form feed and line terminators are defined as *whitespace*.

whitespace	::=	ASCII SP character ($0x20$) "space"
	|	ASCII HT character ($0x09$) "tab"
	|	ASCII FF character ($0x0C$) "form feed"
	|	line_terminator
line_terminator	::=	ASCII LF character ($0x0A$) "line feed"
	|	ASCII CR character ($0x0D$) "carriage return"

Any occurence of whitespace is eliminated during lexical analysis of program compilation to obtain the tokens necessary for syntactical analysis.

Comments

Just as in other popular programming languages such as Java [65] or C# [76], comments in RDL can be of two different kinds: Either, all text in between the ASCII characters /* and */ is ignored, or the characters // indicate that all text following them up to the end of the line is discarded.

comment	::=	single_line_comment
	|	multi_line_comment
single_line_comment	::=	'//' [input_character*] line_terminator
input_character	::=	ASCII input character
		line_terminator
multi_line_comment	::=	'/*' not_star_char comment_tail
comment_tail	::=	'*' comment_tail_star
	|	not_star_char comment_tail
comment_tail_star	::=	/
	|	'*' comment_tail_star
	|	not_star_not_slash_char comment_tail
not_star_char	::=	input_character *except* '*'
	|	line_terminator
not_star_not_slash_char	::=	input_character *except* '*' or '/'
	|	line_terminator

Identifiers

An *identifier* is a sequence of ASCII characters with the restriction that this sequence may neither equal an RDL *keyword*, nor a *boolean_literal*.

identifier ::= [a−zA−Z_][a−zA−Z_\−0−9]∗ *except* keyword or boolean_literal

Keywords

Keywords are character sequences that may not be used as identifiers in RDL programs since they convey a specific meaning for syntactic interpretation.

keyword ::= *one of* 'call' 'count' 'define' 'exists' 'eval' 'fact' 'flush' 'max' 'min' 'name' 'pow' 'retract' 'rule' 'ruleset' 'send' 'set' 'slot' 'sum' 'this' 'touch'

Apart from the above mentioned keywords, a number of reserved system identifiers exist. These system identifiers mainly encapsulate domain-specific aspects of the FACTS middleware framework. Although the same constraints on usage applies to them as to the keywords, they are not part of the language but of the implementation of FACTS, and are therefore discussed in Section 6.3.

Literal

Value domains supported by RDL include integer values, strings and booleans. Literals are the source code representation of these (primitive) types.

literal ::= integer_literal
| string_literal
| boolean_literal

integer_literal ::= [0 − 9]+

string_literal ::= "[a−zA−Z_][a−zA−Z_\−0−9]∗"

boolean_literal ::= 'true'
| 'false'

Due to the target domain of embedded systems and the predominant 16bit architcture of utilized microcontrollers, Integers have a width of 16bit and are read in an unsingned manner. The value range covered by RDL therefore equals to 0 - $(2^{16} - 1)$.

Seperator

To syntactically seperate individual parts of the programming language, a set of symbols is reserved to represent these seperators in RDL. The following tokens are part of this set.

seperator ::= *one of* '[' ']' '{' '}' '(' ')' ',' '←' '→'

Operator

RDL supports the expression of both unary and binary operations on values as well as comparison operations for evaluation. While unary operations are mainly denoted with the help of predefined keywords, the language sticks to commonly utilized symbols for binary expressions. Logical and arthimetical operations are available for processing input values in RDL.

operator ::= *one of* '+' '−' '∗' '/' '%' '&' '|' '^' '∼' '='
'==' '! =' '<' '>' '<=' '>='

5.1.2 The Syntactic Grammar

The specification of a lexical grammar basically enables the implementation of a parser. With this at hand, an input stream of symbols denoted in the source code file can be read and tokenized. Akin to natural languages processing, this transformation maps characters to words.

With its specification of valid combinations of tokens, a syntactic grammar allows for forming sentences from input tokens, which corresponds to the next step necessary towards gaining executable binaries. Implementationwise, a syntactical grammar is needed to write a compiler for a language. The following sections cover the syntactic grammar of RDL and clarify what structure precisely input well-formed RDL programs have to conform to.

Names

The declaration of and reference to entities within a programming language depends on names. Namable entities in RDL are names, facts and slots which are subject to the preceeding section. A concept that is tightly coupled to names is the question of scope, thus the validity and visibility of a name within a program. Within RDL, the scope of a name corresponds to the ruleset that it has been declared in. Since the core language supports only a single, global ruleset, problems

such as shadowing cannot appear. Nevertheless, the availability of naming names allows for clarifying the utilized names within one ruleset.

```
named_name  ::=  'name' identifier '=' name
name        ::=  identifier
             |   string_literal
```

Data Types

The ruleset definition language is a weakly typed language. Variable types are not explicitly declared at initialization time but simply derived by the compiler during compilation from input values. Hence, typing is static, performed at compilation time with the advantage of requiring less effort on the part of the programmer. Nevertheless, the interpretation of RDL is type safe: the runtime environment checks at execution time whether requested operations on values are allowed and handles errors accordingly.

```
type            ::=  primitive_type
                 |   composite_type
                 |   filter_type
primitve_type   ::=  boolean_literal
                 |   integer_literal
                 |   string_literal
composite_type  ::=  fact
filter_type     ::=  slot
```

Unlike widespread programming languages such as C or Java, RDL does neither distinguish Strings from characters, nor support common string operations such as concatenation or substring extraction. For reasons of bytecode conciseness and lack of user interaction at runtime, these features have been omitted. Instead, Strings are mapped to unique integer values during the compilation process and can therefore be categorized as primitive types. This design decision has the advantage that variable values are of constant size, a fact that enables efficient memory management insusceptible to errors at runtime.

Besides primitive data types, RDL specifies a composite_type called *fact* and a filter_type called a *slot*. Each fact has at least a name and may optionally be declared to have a set of properties. These named tuples of key/value pairs constitude the data abstraction available to a programmer. Qualified access to corresponding values can be obtained by refering to the appropriate name and/or property of the fact. Note that there is no declaration phase followed by an

instantiation phase as it is common in object-oriented languages. Actuals are directly assigned to the properties of a fact at declaration time.

fact	::=	name property_list_opt
property_list_opt	::=	['[' property_list ']']
property_list	::=	property { ',' property }
property	::=	key '=' variable
key	::=	[a−zA−Z_][a−zA−Z_\−0−9]∗
variable	::=	boolean_literal
	|	integer_literal
	|	string_literal
	|	slot

Fact processing itself is subject to pattern matching which implies a non-standard programming model for data access: Filtering of facts via slots substitutes for explicit binding of facts to variables or for direct memory access via addressing. A slot specifies the kind of fact and/or fact property that a programmer wants to filter, possibly with a set of conditions that have to furthermore be met to return a match. Hence, slots are neither pointers nor references to specific facts or fact properties but are resolved upon available facts at runtime. This way, the language itself prohibits any dynamic memory allocation on behalf of an application developer besides fact definition.

slot	::=	identifier
	|	'{' name condition_list_opt '}'
	|	'{' name key condition_list_opt '}'
condition_list_opt	::=	['[' condition_list ']']
named_slot	::=	'slot' identifier '=' slot

Slots can be named so that reuse of declared filters within the ruleset is available. An in-depth discussion of condition evaluation will be presented in following subsections.

Blocks

The fundamental execution entity of an RDL program is a ruleset. An identifier has to be assigned to each ruleset, followed by an arbitrary number of blocks of sequences of either declarations of names or slots, definition of facts or execution instructions via rule specifications. Naturally, names of facts and slots referenced within a rule have to be declared beforehand to ensure proper use, a property that can easily be verified by the compiler.

```
ruleset      ::=  'ruleset' identifier block_list
block_list   ::=  block {block}
block        ::=  named_name
                | named_slot
                | fact
                | rule
```

The definition of facts within a block is comparable to the definition of static variables common in programming languages such as C or Java. Hence, they mainly serve two different objectives: First of all, these facts store constants used throughout the ruleset, e.g. parameters for system-related configuration or thresholds for measurements. Then, the programmer should not evict them during runtime to assure a running system. The second possibility is to use them in a volatile manner to initialize e.g. facts produced at runtime or parameterize initialization calls to underlying system functions such as timers or sensors. In this case, cleaning up the facts after usage will free precious resources at runtime.

All facts declared ouside a rule context will be instantiated once when program execution begins.

Rules, Conditions and Statements

Syntactically, rules in RDL follow a simple IF condition THEN action structure. Each rule is tagged with a specific identifier, a priority to indicate execution order in compliance with other rules, a list of conditions that correspond to the IF part and a set of statements that denote the action part.

A rule fires whenever the conjunction of all its conditions evaluates to true. Then, the set of statements will sequentionally be executed in the order of their definition in an atomical manner. Unlike other languages that are popular especially in the ECA domain, RDL offers no ELSE construct. Nevertheless, the semantics of RDL rule triggering requires a fresh event, thus a newly added or altered fact to take part in the condition evaluation, which will be discussed in detail in Section 5.2.

```
rule             ::=  'rule' identifier priority condition_list
                      statement_list
priority         ::=  [-] [0-9]+
condition_list   ::=  '←' condition [condition_list]
statement_list   ::=  '→' statement [statement_list]
```

Conditions allow to specify guards upon actions. Since all data is represented in the format of a fact, conditions consequently probe whether facts exist or value ranges of their properties can be met. Two different kinds of conditions can be defined:

With the help of a condition starting with the *exists* keyword, a programmer can test upon existance of a specific fact, possibly requesting it to satisfy given constraints. An *eval*-condition allows to compare values of fact properties to one another with a multitude of operators. Naturally, conditions can be nested to obtain a suitable expressivity for the given problem domain.

```
condition       ::=  'exists' slot
                   | 'eval' '(' expression comparison_op expression ')'
comparison_op   ::=  '=='
                   | '! ='
                   | '>'
                   | '<'
                   | '>='
                   | '<='
```

Statements encapsulate the actual instructions for processing facts. Available operations are straightforward and enable a developer to insert new facts into the fact repository with the *define* statement, to delete facts matching a specified filter with the *retract* statement, to update individual properties of a fact via the *set* statement, to send a fact either in a broadcast or unicast manner to neighboring nodes via the *send* statement, to mark facts as unmodified or modified or to call a system function.

```
statement   ::=  'define' name initializer_list_opt
               | 'retract' slot
               | 'set' slot '=' expression
               | 'send' expression expression slot
               | 'flush' slot
               | 'touch' slot
               | 'call' identifier expression_list_opt
```

When defining a fact during rule execution, its properties can be assigned using a list of initializers. The rvalues of each property are set to equal an expression, whose value can be derived at runtime, see also the following paragraph.

```
initializer_list_opt   ::=  [ '[' initializer_list ']' ]
initializer_list       ::=  initializer ',' initializer
initializer            ::=  key '=' expression
```

Expressions

Expressions capture most of the work done during rule processing. Used for denoting side effects including assignment of values to fact properties, for evaluation of filtering conditions to determine rule triggering or for denoting values to be passed as arguments, expressions are the central tool to control the values of fact properties.

```
expression_list_opt  ::=  [ '('expression_list')' ]
expression_list      ::=  expression { ',' expression}
expression           ::=  variable
                        |  '(' unary_op expression ')'
                        |  '(' expression binary_op expression ')'
```

To be able to support a multitude of operations on fact properties conveniently, RDL offers a rich set of operators. Unary operations include the possibility to count the number of available facts of a certain kind, to sum up the values of a dedicated property of all facts of a certain kind, to filter the minimum or maximum value of a property, and to filter the negation of a specified value of a property. Naturally, a concatentaion of filtering conditions on the facts in question can furthermore be applied.

Binary operations include addition, subtraction, product, division, the modulo and the exponential operation, as well as the logical operations AND, OR and XOR.

```
unary_op   ::=  'count'
            |   'sum'
            |   'min'
            |   'max'
            |   '~'
binary_op  ::=  '+'
            |   '-'
            |   '*'
            |   '/'
            |   '%'
            |   'pow'
            |   '&'
            |   '|'
            |   '^'
```

5.2 A Denotational Semantics for RDL

The formal specification of the semantics of a language is an important step to warrant its correct implementation across platforms. One way to fulfill this demand is to specify its *operational semantics*: The evaluation process of a program written in the language being fed into an interpreter then denotes its meaning. The Haskell interpreter [136] which has been designed for RDL is an instance of such an operational approach. Although in many cases an algorithmic representation such as the implementation of an interpreter can give a valuable intuition towards how the language is to be perceived, machine-independence cannot be provided since language semantics take the form of interpreter configurations.

The provision of *axiomatic semantics* is another common method to illustrate intended language semantics. Symbolic logic is used to define properties of language constructs as axioms and inference rules so that program properties can afterwards be deduced. This approach is especially suited to point out characteristic properties to a programmer and prove their validity, but may run short on actually capturing a complete program‘s meaning.

A third option towards the specification of language semantics which lies in between the above mentioned ends is to provide what is called the *denotational semantics* of the language in question. The goal here is to construct a meaning function that produces the (input/output) function of a program, given any program in the corresponding language [132, 126]. In the following, we will establish the denotational semantics for RDL. To ensure readability, we depend on the formalism introduced for the Starburst production rule system [152] where applicable, but naturally extend it to meet RDL semantics.

5.2.1 Domains

Semantic domains are sets of value spaces in programming languages. As such, they serve as the foundation for functions that operate upon these domains, which in turn are used to specify the meaning function of a specific language. RDL comprises the following domains:

- Let FS be the domain of *fact repository states*.

  If f is a state in FS, then $f = \{f_1, f_2, ..., f_n\}$, with each f_k being a *fact*, which is a compound set of named tuples, called *properties*.

- Let P be the domain of *well-formed properties*. Let f_k be a fact in f, then $f_k = \{f_k.p_j, f_k.p_{id}, f_k.p_{name}, f_k.p_{time}, f_k.p_{state}\}$ denotes its properties with $j \geq 0$ and where each property p_i is a key-value tuple. We assume that facts are automatically tagged with a set of four different system properties (p_{system} in the following) at the time of their creation. These are, in contrast to the regular properties, mandatory for every fact and include an identifier naming the last node that altered the fact, the name of the fact, a timestamp identifying its creation or update time and its current state.

- Let Δ be the domain of *sets of fact repository changes*.

  If δ is a set of changes in Δ, then $\delta = [M, Ret, E, Rep]$, with M denoting the set of *modifications*, Ret the set of *retractions*, E the set of *events* and Rep the set of *repressions*. For the sake of simplicity, let δ_m identify solely the set of modifications, thus $\delta_m = [M, \emptyset, \emptyset, \emptyset]$, let δ_{ret} denote the set of retractions, thus $\delta_{ret} = [\emptyset, R, \emptyset, \emptyset]$, let δ_e denote the set of events, thus $\delta_e = [\emptyset, \emptyset, E, \emptyset]$ and finally let δ_{rep} denote the set of repressions, thus $\delta_{rep} = [\emptyset, \emptyset, \emptyset, Rep]$.

  $M = \{\langle f_i.p_j, v_j\rangle\}$ where $1 \leq i \leq n$ and $\langle f_i.p_j, v_j\rangle$ exists. Each p_j is a property of a manipulated fact f_i and v_j is the value of the corresponding property. M subsumes all facts that have been newly defined due to rule execution or whose property values have been changed. Consequently, these facts are results of calling the *Define* or the *Set* statement in the rule action part, see Section 5.1.2.

  $Ret = \{f_i\}$ where $1 \leq i \leq n$ and each f_i is a fact that has been retracted from the fact repository. Ret thus subsumes all facts that are not available any more after the execution of a rule because they have been deleted by calling the *Retract* statement. Evidently, a fact is retracted with all its properties.

  $E = \{f_i\}$ where $1 \leq i \leq n$ and each f_i is a fact that represents an event in the fact repository. Facts in E have either been explicitly set to appear as events via a call of the *Touch* statement at execution time, or they have been added by the runtime environment to the repository after the production process returned.

  $Rep = \{f_i\}$ where $1 \leq i \leq n$ and each f_i is a fact whose ability to trigger other rules is repressed. Facts in Rep have been modified to repress reactions by means of calling the *Flush* statement within the action part of a rule.

  Note that in the current implementation of the FACTS runtime, facts are assigned to belong to the above mentioned sets of changes by simply tagging them with specific system state variables.

- Let R be the domain of *RDL production rules*.

  If r is a rule in R, then generally r is a function that takes a set of fact repository changes δ and a fact repository state f as input parameters, to then return a boolean value, a new fact repository state as well as a new set of changes. Thus,

  $$r : \Delta \times FS \rightarrow \{true, false\} \times \Delta \times FS$$

  Let C be the domain of *sets of RDL rule conditions* and A be the domain of *sets of RDL rule actions*. An alternative representation for a rule r_i is to split it into a concatenation of its conditions and actions. This way, we are able to be more specific about the conditions to be met for rule triggering than e.g. in [152].

If c_i^k is the set of k conditions in C for rule r_i, then c is a function that takes a set of fact repository changes δ and a fact repository state f as input parameters to return a boolean value. In the following, let f_{c_i} denote the set of facts that are referenced in c_i^k. Then, the set of conditions of rule r_i evaluates to true in case all of the following holds:

$$c : \Delta \times FS \rightarrow \{true, false\}$$
$$c_i(\delta, f) = true \text{ iff } \bigwedge_{k=1}^{n} c_i^k = true$$
$$\wedge\, \exists f_j \in f_{c_i}, \text{ so that } f_j \in \delta_e.$$

Thus, the conjunction of all conditions of the corresponding rule have to be true and at least a fact involved in successful condition evaluation has to be an event.

Accordingly, a set of actions a_i^l in A will be triggered only in case c_i returns $true$, and will then take the set of changes δ and fact repository state f to return new changes δ and a new state f.

The set of actions is denoted as follows[1]:

$$a : \Delta \times FS \rightarrow \Delta \times F$$
$$a_i^l(\delta, f) \downarrow 1 = [\emptyset, \emptyset, \emptyset, \emptyset] \text{ and } a_i^l(\delta, f) \downarrow 2 = f \qquad \text{iff } c_i(\delta, f) = false$$
$$a_i^l(\delta, f) \downarrow 1 = [M', Ret', E', Rep'] \text{ and } a_i^l(\delta, f) \downarrow 2 = f' \text{ iff } c_i(\delta, f) = true$$

meaning that neither an operation is scheduled, nor a new state is obtained in case the rule condition is $false$. Condition evaluation is thus free of side effects. Otherwise, the actions a_i^l of r_i are executed, which yields a new fact repository state as well as a new set of changes, see also function *Accumulate-Change* in the next section.

- Let O be the domain of *priority orderings of rules*.

  If o is a priority ordering in O for rules r in R, then
  $o = \{r_j \geq r_i, \mid r_j \text{ has precedence over } r_i\}$ with $1 \leq i, j \leq n$ and where $\geq$ is transitive and irreflexive, but not necessarily total. Without loss of generality, we assume the index i of a rule r_i to be expressed as a natural number, thus $i \in \mathbb{N}$ with a greater index indicating a higher scheduling priority.

- Let PS be the domain of *sets of processing states*.

  If σ is a set of processing states in PS, then $\sigma = [Av, Ex, i]$ with $Av \subseteq P(R)$ denoting the set of rules available for execution, $Ex \subseteq P(R)$ describing the set of rules that have been executed during the current production run and $i \in \mathbb{N}$ denoting the index, thus scheduling priority of the last rule that has been executed. Note that $P(R)$ identifies the *powerset* of input rules. Once

[1] As introduced in [152], we use $\downarrow i$ to denote the projection on the ith element in a Cartesian product.

again for the sake of simplified denotation, we will refer with σ_{av} solely to the set of available rules, σ_{ex} to the set of executed rules and and σ_i to the index.

Then, $\sigma = [\{r_{av_j}\}, \{r_{ex_k}\}, i]$ so that $0 \leq av_j, ex_k \leq n$ and $Av \cap Ex = \emptyset$ and $1 \leq i \leq n$.

5.2.2 Supporting Functions

In the following, all formal definitions of functions are given using the widespread λ-calculus, originally introduced by Church [80], which allows to express a function's actions on its arguments. A number of input expressions E_i, each separated by a comma with $1 \leq i \leq n$, is simply mapped to a (set of) output expressions, which in turn are separated by a period from the input. The meaning function that actually captures language semantics depends on a concatenation and composition of the following auxiliary functions.

- Function *Get-Index* takes a rule r_i and returns its index, which represents its scheduling priority.

  Get-Index: $R \rightarrow \mathbb{N}$

  Get-Index $= \lambda\ r_i.\ i$

- Function *Max-Priority* takes a set of rules (with $P(R)$ being the powerset of rules) and an ordering O and returns the maximum index of the rules, which corresponds to the index of the rule with the highest scheduling priority.

  Max-Priority: $P(R) \times O \rightarrow \mathbb{N}$

  Max-Priority $= \lambda\ \{r_1, r_2, ..., r_n\}, o.$

  $$\textit{Get-Index}(r_i) \text{ so that } \forall i \neq j,\ 1 \leq j \leq n : r_j > r_i \notin o$$

- Function *Update* takes a rule r_i being executed and a processing state σ and returns a new processing state that will be valid after rule execution.

  Update: $R \times PS \rightarrow PS$

  Update $= \lambda\ r_i, [\{r_{av_j}\}, \{r_{ex_k}\}, i].$

  $$[\{r_{av_j}\} \setminus r_i, \{r_{ex_k}\} \cup r_i, \text{Get-Index}(r_i)]$$

- Function *Run-Rule* takes a rule r_i, a fact repository state f, a processing state σ, a set of changes δ and a processing state σ. It returns
  (1) the new state of the fact repository obtained after executing all of r_i's actions, starting with the fact repository state f,
  (2) the global set of changes resulting from accumulating the new set of changes produced by executing r_i to the set of changes that have been obtained from previously executed rules and
  (3) a new processing state for the current run.

Run-Rule: $R \times FS \times \Delta \times PS \rightarrow FS \times \Delta \times PS$

Run-Rule $= \lambda\ r_i, f, \delta\sigma.$

$$\langle r_i(\delta, f) \downarrow 3,\ \textit{Accumulate-Change}(r_i(\delta, f) \downarrow 2,\ \delta),\ \text{Update}(r_i, \sigma)\rangle$$

- Function *Accumulate-Change* takes two sets of changes δ_1 and δ_2 and returns the set of accumulated change, also denoted as the *net effect* of change. For instance, a modification or repression of a fact will render no change in case it is deleted afterwards.

  Accumulate-Change: $\Delta \times \Delta \rightarrow \Delta$

  Accumulate-Change $= \lambda\ [M_1, R_1, E_1, Rep_1],\ [M_2, R_2, E_2, Rep_2].$

  $[M', R', E', Rep']$, with

  $M' = (M_1 -^* (R_2 \cup E_2 \cup Rep_2)) \cup M_2$

  $R' = R_1 \cup (R_2 -^* (M_1 \cup E_1 \cup Rep_1))$

  $E' = (E_1 -^* (M_2 \cup R_2 \cup Rep_2)) \cup E_2$

  $Rep' = (Rep_1 -^* (M_2 \cup R_2) \cup E_2) \cup Rep_2$

  where $F_1 -^* F_2$ is defined as $\{f_x \in F_1 \mid f_x \text{ does not appear in } F_2\}$.

- Function *Swap* takes the net effect of the sets of changes produced during a run of the rule engine through all input rules and recalculates a new set of event changes δ_e. Any modifications to the fact repository are therefore simply swapped to be new events and thus allow for a new production run. All other sets of changes are omitted since they are irrelevant for future rule scheduling decisions.

  Swap: $\Delta \rightarrow \Delta$

  Swap $= \lambda\ [M, R, E, Rep].\ [\emptyset, \emptyset, M, \emptyset]$

- Function *Prepare-Production* takes the global set of changes, a processing state and a priority ordering to re-assign all settings necessary for a new production run through the corresponding rules. Therefore, it returns a revised version of the sets of changes and initializes the processing state to contain all rules in the set of available rules, none in the set of executed ones and finally sets the index to the rule with the maximal scheduling priority. This will be the starting point for a new rule evaluation run.

  Prepare-Production: $\Delta \times PS \times O \rightarrow \Delta \times PS$

  Prepare-Production $= \lambda\ \delta, \sigma, o.$

  $$\langle\ \textit{Swap}(\delta), [\{r_1, ..., r_n\}, \emptyset,\ \textit{Max-Priority}(\{r_1, ..., r_n\}, o)]\rangle$$

- Function *Choose-Triggered* takes a processing state, the global set of changes, a fact repository state and a priority ordering. It returns a rule r which is triggered by the changes present in δ such that no rule with precedence over r in o is triggered.

Choose-Triggered: $PS \times \Delta \times FS \times O \rightarrow R$

Choose-Triggered $= \lambda\ \sigma, \delta, f, o.$

$$Select(Eligible(\sigma, \delta, f, o))$$

- Function *Eligible* takes a processing state σ, the changes δ, a fact repository state f and a rule ordering o. As a result, it return a set of rules that are triggered by the changes and according to the current state of the fact repository and the current processing state such that no other rule in r with precedence over the chosen one is present in o.

  Eligible: $PS \times \Delta \times FS \times O \rightarrow P(R)$

  Eligible $= \lambda\ \sigma, \delta, f, o.$

  $$\{r_i \mid 1 \leq i \leq \sigma_i\ \wedge r_i \in \sigma_{av} \wedge\ r_i(\delta, f) \downarrow 1 = true\ \wedge$$
  $$\{r_j \mid 1 \leq j \leq \sigma_i \wedge r_j \in \sigma av \wedge r_j(\delta, f) \downarrow 1 = true\ \wedge$$
  $$r_j > r_i \in o\} = \emptyset\}$$

- Function *Select* takes a set of rules and deterministically chooses one.

  Select: $P(R) \rightarrow R$

  Select is undefined and never applied to the empty set. As we assume a programmer to be aware of race conditions when explicitly specifying rules of the same priority, we take as given a reasonable definition of this function.

5.2.3 The Meaning Function

Semantics of programming languages are usually denoted by a function commonly referred to as the *meaning function* $\mathcal{M}$. Input to $\mathcal{M}$ are the set of rules $R \in P(R)$ and an ordering $o \in O$, given that all rules in o are also in R. $\mathcal{M}[R, o]$, thus the meaning of R and o, is a function operating on a set of changes δ and a fact repository state f. It eventually outputs a new fact repository state which is a result of processing the rules according to the given input. In case rule processing does not terminate, the function returns $\perp$ (bottom). $\mathcal{M}$ is defined as follows:

$$\mathcal{M} : P(R) \times O \rightarrow \Delta \times FS \rightarrow FS \cup \{\perp\}$$

$$\mathcal{M}[\{r_1, ..., r_n\}, o]\ =\ \lambda\ \delta, f.\ f.\mathcal{M}'(f,\ Prepare\text{-}Reaction(\delta, \{r_1, ...r_n\}, o))$$

Function *Prepare-Reaction* takes a set of changes, thus afore recognized events, a set of rules and a priority ordering to start checking for denoted reactions to these events. It simply returns the set of changes, the set of rules and the index of the rule(s) with the highest priority to be evaluated first, which is then used to bootstrap the rule evaluation process. Note that events that have occurred during a production rule run are automatically inserted into δ by the rule engine when these productions have terminated.

$$\textit{Prepare-Reaction}\colon \Delta \times P(R) \times O \rightarrow \Delta \times PS$$

$$\textit{Prepare-Reaction} = \lambda\ \delta, \{r_1, ..., r_n\}, o.$$

$$\langle [\emptyset, \emptyset, E, \emptyset], [\{r_1, ..., r_n\}, \emptyset,\ \textit{Max-Priority}(\{r_1, ..., r_n\}, o)] \rangle$$

The final recursion is captured in function $\mathcal{M}'$, which takes a rule ordering o to return the least fixed point function F. Input parameters to F are a fact repository state f, a set of changes δ and a processing state σ. In case no (more) rules are triggered by the changes in δ, F returns a (potentially new) fact repository state f. In all other cases, it calls the function *Choose-Triggered* to choose a rule r_i with maximal priority in regard to the current processing state, and re-applies itself to the new state, changes and processing state which result from calling *Run-Rule*. Eventually, when no rule is eligible to be processed in respect to available rules in the processing state, all modifications to the fact repository will be scheduled as new events calling the function *Prepare-Production* and rule evaluation restarts.

$$\mathcal{M}' : O \rightarrow FS \times \Delta \times PS \rightarrow FS$$

$$\begin{aligned}
\mathcal{M}' \ = \ & \lambda\ o.\textit{Least-Fixed-Point}(\ \lambda\ F. \\
& \lambda\ \langle f, \delta, \{r_1, ..., r_n\}, \sigma \rangle. \\
& \quad \text{if } \textit{Eligible}(\sigma, \delta, f, o) \ = \ \emptyset \wedge \delta_m = \emptyset \text{ then } f \\
& \quad \text{else if } \textit{Eligible}(\sigma, \delta, f, o) \ = \ \emptyset \wedge \delta_m \neq \emptyset \\
& \quad\ \ \text{then } \textit{Prepare-Production}(\delta, \sigma, o) \\
& \quad \text{let } r_i = \textit{Choose-Triggered}(\sigma, \delta, f, o) \text{ in} \\
& \quad\ \ F(\textit{Run-Rule}(r_i, f, \delta)))
\end{aligned}$$

As can be seen, the actual process of rule evaluation is split into two, alternating phases: In a first phase, all reactions to the current events in δ_e are scheduled for execution according to the rule processing order o. As soon as all appropriate rules have been scheduled, the set of changes is updated by omitting all events, retractions and repressions, setting all modifications to function as new events and once again resume processing by re-entering the first phase.

5.2.4 Discussion

The formal notation of the declarative semantics presented above have clearly been inspired by the work presented in [152]. Since both, the Starburst production rule language and RDL, share common concepts such as rules, sets of changes that influence rule scheduling decisions and a current state for the data, the corresponding definitions of these domains and the naming of supporting functions have been chosen to match where appropriate to not confuse the reader. However, fundamental differences regarding the nature of the sets of changes and the effect of change on the scheduling decisions are worth pointing out.

The Starburst rule language has been specified to operate in a database context. Production rules serve as a means to denote active behavior, triggered by classical database operations such as *INSERT, DELETE*, and *UPDATE* operations and are applied to specific tables within the database. Rules are activated for instance at the end of a successful transaction or at any other user-specified *rule assertion point*, but not at an arbitrary point in time. Upon activation, the sets of changes are distributed to all rules and the rule with the highest priority amongst all activated rules is chosen and executed in case the conditions are met. All changes that result from this execution are directly re-applied to the former set of changes, re-distributed to all rules, and once again, the activated rule with the highest priority is a candidate for scheduling. The effect of change due to rule execution is therefore immediately visible.

The RDL rule language favors a different execution semantics for its rules. First of all, rule processing will start at an arbitrary point in time, namely whenever a new event has been recognized. The function *Prepare-Reaction* expresses exactly this reactive behavior as it triggers a new run through the available rule base upon the occurrence of a new element in δ_e. Data sample facts, timer interrupt facts or facts received via the radio transceiver pushed into the fact repository by the runtime environment are typical events that appear in δ_e. Reaction granularity is therefore on the level of tuples or facts, not on tables and favors an immediate rule execution model. Naturally, this is a result of the envisioned application domain of event processing rather than to utilize RDL for data monitoring.

Due to a different data manipulation schema and execution model for rules supported by the RDL language, the available sets of changes as well as their influence on the scheduling semantics tremendously differ from those available within the Starburst system: Rule evaluation is divided into individual runs, which correspond to a complete, in-order evaluation cycle of the rule base. Upon event recognition, the rule with the highest priority reacting to this event is scheduled for execution and, in case the actions modify the fact repository, the incurred change is added to δ by calling the *Accumulate-Change* function. Note however that fact insertion or updates are *not* automatically visible as events, but contribute to the modifications in δ_m, instead. Afterwards, the run resumes, evaluating all rules with lesser or equal priority than the formerly executed rule, see function *Eligible*, thus preserving rule evaluation order within a run, until no more rule is eligible to trigger. Consequently, one event may trigger multiple rules unless the corresponding fact is removed prior to the evaluation of subsequent rule premises.

In case no more rule is eligible to trigger, two possible operational paths can be followed: Either, no modifications to the fact repository have occurred during the previous run, thus the productions have come to a stable state and the rule evaluation process terminates. Or, the net effect of change regarding fact modifications is not empty, thus new facts have been produced or facts have been altered, leading to the necessity to resume a new production rule run. Then, all old events δ_e which have been properly reacted to during the last run are evicted from the set of changes, and all modifications are instead swapped to serve as new events that

possibly demand for reactions by calling the function *Prepare-Production*. Once again, recursive evaluation of the rule base is triggered.

Note that the differentiation of facts into modifications and events has a clear advantage when utilizing production rules in an event-processing domain, since it fuses production rule *and* ECA rule semantics into one, hybrid concept. As the syntax suggests, the programmer is facing production rules realizing stateful semantics: Fact insertion, alteration and retraction directly influence the set of rules that may trigger immediately after the corresponding rule has been executed. The conflict set of rules is recalculated as fact repository state may have changed, and as a result rules may join or leave this set. However, rule evaluation order is kept and the event set of changes is not effected unless *on purpose*.[2] Rules can thus generate new events, but not necessarily every fact that e.g. serves as a temporary variable, a means to store stateful information or to pass data between rules automatically becomes an event. From a programming perspective, reaction and production are manageable in an easy manner at the same time with the same data abstraction.

5.3 RDL Language Pragmatics

Given the fact that a programming language is the vehicle to map a problem of a certain domain or application area, a specific routine or process to software, the pragmatics of a language can be understood as an indication for how well the chosen language suits the demand for expressiveness and applicability. In order to judge, or more precisely investigate language pragmatics, the following section will shed some light on common programming patterns which result from extensive work on and with RDL. Note that the intention is to discuss language usage at a rather abstract level to provide an understanding of general issues rather than application-specific problems.

RDL is a language that has been designed specifically to meet challenges exposed by the wireless sensor network domain, but at the time is general enough to serve as a holistic programming language in a sense that it does not restrict a programmer to a certain class of implementable applications or protocols for networks. In order to achieve both of the stated goals, simple event processing routines as well as what can be labeled as *algorithmic knowledge* have to be expressible in RDL rules. As has been stated before, the usage of reactive production rules in a sensor networking context is beneficial since it nicely leverages event-centric processing *and* knowledge representation and combines them in a single mental model.

The patterns presented below can be roughly categorized into two distinct groups: The first class comprises those patterns that are applicable to overcome problems which result from choosing a rule-based abstraction in the first place: since control flow is completely shielded from a programmer and handed to a rule

[2] This can be the case when implementing control sequences, see 5.3.

engine, one has to literally trick the system in case the implementation relies on a more sophisticated control mechanism than simple if-then-else statements. It is important to point out, that RDL has *not* been specified to primarily focus on procedural programming, since then, choosing a rule-based programming abstraction would have been simply wrong. Rather, the common case is an event-based processing scheme, and the intention of presenting control patterns is to point out suitable workarounds for those cases that do need special attention.

The second group of patterns are processing schemes specific to the sensor networking domain. These are simply general observations how to deal with incoming data in RDL in the most efficient way, how to organize common data processing steps and how to speed up rule evaluation by minimizing the fact repository search space.

5.3.1 Control Patterns

Imperative programming languages are nowadays prevalently used. As a consequence, a software developer is prone to structure and express algorithms in terms of basic control structures he/she is familiar with. Therefore, languages that lack these control mechanisms can lead to mild irritations since the programmer has to rethink the problem. For instance, popular control structures such as `while` and `for` loops can be essential to capture processing semantics, but have to be mapped to consecutive event-action pairs to be utilized in a rule-based implementation. Unlike execution strategies for imperative languages where execution state is automatically obtained in the program stack, event-centric processing lacks a governed and for a programmer accessible state. Therefore, state has to be explicitly managed by the application programmer, a necessity that one has to be aware of. A third mechanism that is lacking is the ability to outsource common functionality applicable to a variety of data items to a method or function.

Control patterns, such as the ones presented below, are always concerned with *node-local* processing to allow for interaction and information passing between individual rules and thus can be understood as enablers for sophisticated data processing algorithms.

The WHILE-Loop

A common problem a developer encounters when having to express iterations in rule-based languages such as RDL is that there is a need to map this behavior into interacting event-action pairs and to manually control the looping conditions. Therefore, to implement a `while` loop in RDL, one has to take advantage of the fact that (1) production rule evaluation will not stop unless no new events appear in the fact repository during the last run and (2) each production run itself can be understood as a loop.

Recall once again the definition of successful condition evaluation from Section 5.2.1: A rule will be executed if all of its conditions evaluate to true, and one

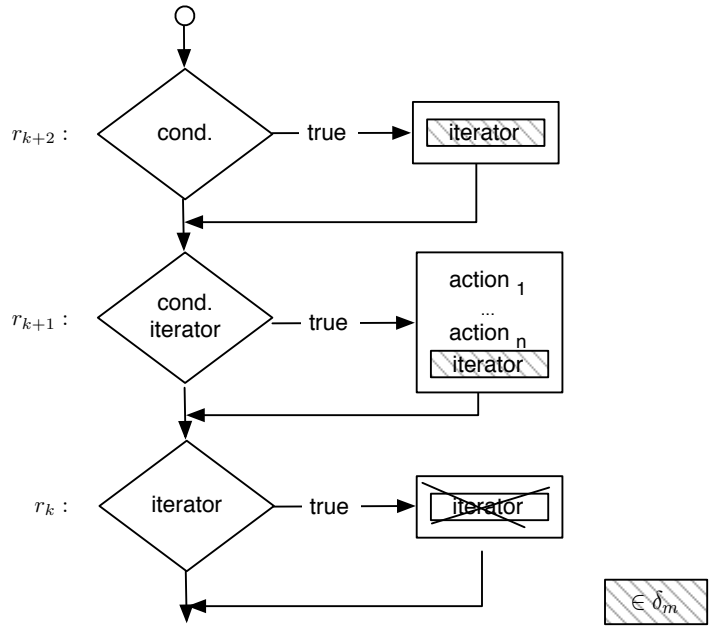


Figure 5.1: Mapping a `while` loop to RDL rules

of the facts involved in successful condition evaluation is tagged to be an event via its state property. Thus, to make use of the repetitive rule evaluation process, a fact that controls the `while` statement and is kept to appear as an event, has to be generated. One could also think of this as a `GOTO` statement that operates across production rule runs.

Figure 5.1 depicts the idea of incorporating iterative processing into RDL. The key is to simply split the task of loop control and condition evaluation into two separate concerns, thus spawn a new fact (*iterator* in this particular case) when the `while` condition (denoted as *cond* in the flow chart) becomes true. Then, execution of every statement in the body of a traditional `while` loop is guarded by both, the `while` condition and the iterator fact. This ensures that the body of the loop will be evaluated and eventually executed in a *sequential* manner, an execution path that is unusual for rule-based processing, but mandatory for correct loop reproduction. At the end of the body, the fact for loop control has to be updated, no matter whether a true update of its values is necessary or not. This warrants that the fact will be swapped into the set of modifactions δ_m and re-appears to be an event once a new production run is scheduled. The last rule r_k discards the loop control fact and will only fire in case its preceding rule did not. While this rule is not mandatory to implement a `while` loop, it is however good programming practice to not thrash the fact repository. Naturally, any rule that has a lesser priority than r_k has to be inhibited to fire during the execution

of the conceptual while loop.

An alternative solution to implement the `while` loop would be to update a fact that is referenced in the `while` condition itself. While it may work, experience has shown that this practice is prone to errors, since these facts may well be part of other rule conditions. In this case, unintentional triggering of non-related rules can cause misbehavior of the complete ruleset. Note that the extension to building a `for` loop using the above mentioned pattern is uncomplicated: The fact that controls the loop has to created with an additional property that serves as a counter variable and will be incremented during the update in r_{k+1}. Likewise, the condition referencing this fact for guarding the body of the `while` loop has to be adjusted accordingly.

Finite State Machines

The lack of a call stack, inherent to event-centric processing, is a challenge a programmer has to face when utilizing RDL. Although unfamiliar at first, this problem can be fixed fairly easy when implementing stateful algorithms with the help of a finite state machine. Clearly, the task of managing state is shifted from the system to the programmer which is a burden, however its translation to rules is straight-forward. Moreover, explicit state management clarifies intended program flow, thus improves program readability.

To incorporate a finite state machine into a set of rules, it has been proven useful to establish one dedicated fact for controlling state, see Figure 5.2. This fact (named *state* here) should have a property which is used to reflect the state that the system currently resides in. State transition rules can be specified to fire either in case an arbitrary condition evaluates to true, e.g. as shown in rule r_{k+j}, dependent on a particular state or due to a combination of both, e.g. rule r_k. Naturally, entering a specific state can also lead to the execution of a number of actions, depicted in rule r_{k+i}, that do not incorporate any change to the current state. Be aware that the definition or modification of the state fact will render it to be part of the set of modifications δ_m and that reactions can hence not directly trigger subsequent rules. Instead, they will be delayed to the next production run to prevent race conditions and ensure correct execution order according to the priority ordering.

Once again, a programmer has to be careful to prevent incidental rule triggering which may result from updating the state fact. Here, a rule of thumb is to proceed as follows: In case immediate, unconditional reactions are the only result of the execution of a state transition rule, thus no rules reference the state fact in combination with other conditions in their condition part, the finite state machine can be understood as a generalization of a `GOTO` statement and there is no need to worry about subsequent rule triggering.

However, in case conditional execution during specific states is given, e.g. as denoted in rule r_k, the programmer should carefully check whether its execution is intended to happen in both cases, when reaching the proper state but also

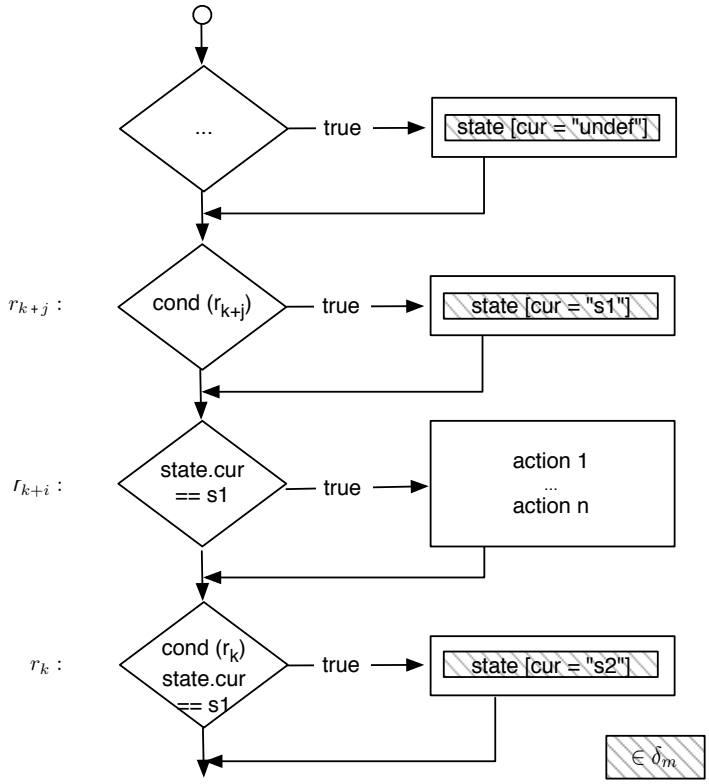


Figure 5.2: Building a finite state machine in RDL rules

when other facts have been modified that are referenced in the condition. To prevent the state fact to act as a trigger, it can always be flushed after state transition. As a consequence, the developer then has to once again split the task of representing state and trigger provision into two separate facts: the state fact, which has to be flushed after every modification and a dedicated trigger fact, e.g. called *state_trigger*, which can be referenced in the condition part of only the corresponding subset of rules that actually are meant to immediately fire.

Generic Matching

The definition of subroutines is a very powerful and basic instrument to organize software: Common tasks that are utilized throughout a program are wrapped into distinct functions, which are then callable from any place in the given source. Since the sourcecode itself is more compact and code duplication avoided due to code reuse, the actual cost of software development can be cut while at the same time maintainability, quality and reliability increase.

A feature that is especially useful in this context is the introduction of polymor-

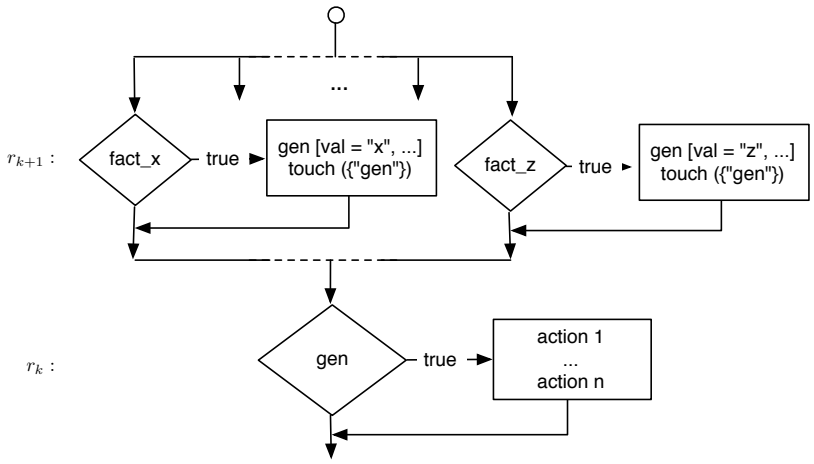


Figure 5.3: Avoidance of code obfuscation with generic matching.

phic functions: Given different types of data that all request for similar processing semantics, the name for calling the corresponding function will be the same, but return value or input parameters may be of different type.

Since control flow is per se not sequential in rule-based processing, code reuse cannot be obtained at the level of subroutines, but has to be treated at the level of proper rule invocation. A problem that appears often in this context is that either different facts are supposed to trigger the same action or that a similar processing scheme is requested to be applied to different facts. Of course this issue shares some resemblance with polymorphic functions, especially when rules serve as guards e.g. to a set of rules that implement a sequential portion of software. However, in this particular paradigm, the quest for genericness may better be expressed as an urge for generic matching capabilities.

We refer to the workaround offered to avoid code obfuscation as *generic matching* in the following.

The problem to solve is to enable successful matching of different fact types against the same filter or condition within language bounds. As fact types are denoted by the name of a fact, which in turn are used to resolve pattern matching requests, the options are to either extend the language or to modify the facts requested to match the filtering condition to the required fact type. The first solution, e.g. realizable via introduction of generic fact types that a fact instance can inherit, is not feasible within RDL, so that the pattern presented here implements a means of casting a fact to another fact type instead. In the exemplary case illustrated in Figure 5.3, this new fact is labeled to be of arbitrary type *gen (for generic)*.

Say for instance that the rule r_k implements a series of actions and/or controls a certain processing loop that should be triggered by or invoked upon different fact

types, *fact_x* to *fact_z* in this case. At first glance, a developer would therefore denote basically the same processing steps (those shielded by the condition of r_k) once for every fact type, each in a closed set of interacting rules, leading to massive redundancy. A more condensed way to achieve the same behavior is to specify one rule per fact type which copies its contents to another fact whose name (thus type) corresponds to the name requested by the actual rule implementing the processing scheme. This generic fact holds all of the properties of its originating fact in addition to a property that stores the fact's name to enable to re-build the source fact after the generic has been handled appropriately. One can understand the generic fact to serve as the trigger *and* the argument to be "passed" for the guarded operations. Type casting is hence implemented by means of fact copying.

The invocation of the `touch` statement upon *gen* is a means to directly influence rule engine control flow. Recall that any new or altered fact will be part of the set of modifications, thus their ability to trigger rule execution is delayed to the next production run. A *touched* fact appears as an event right away - and can trigger rules of lesser priority right away.

Note that in case fact properties are altered or otherwise used in rule r_k, it is mandatory that the different fact types being processed adhere to the same set of properties to warrant proper handling.

5.3.2 WSN-specific Programming Patterns

Event-driven architectures rely on incoming data, or more specifically recognized events, to trigger processing activities. Since they are deployed to acquire, process and eventually share data samples, wireless sensor nodes are a typical representative of such event-driven systems. Independent of the actual application that is implemented, the foremost task of a wireless sensor node is therefore to react to any incoming data in an appropriate manner. The programming patterns introduced below reflect our experience in how to efficiently implement data processing routines in RDL, dependent on the intended application behavior and processing model.

Chain of Filters

The chain of filters pattern is a straight-forward data processing scheme whose intention is the efficient utilization of the scarce resources of a wireless sensor node. To achieve prompt reactivity on every node, processing time as well as allocated facts in the fact repository have to be cut to a minimum. The key to achieving this goal is to simply drop any data item, any fact received via the radio interface, literally any fact residing in the fact repository as soon as possible.

The implementation of a chain of filters makes use of the priority ordering specified for every ruleset. By prioritizing those filtering rules that are very restrictive in terms of data items that pass these rules without being processed and/or retracted, the most common cases are to be filtered first, see Figure 5.4. Subsequent

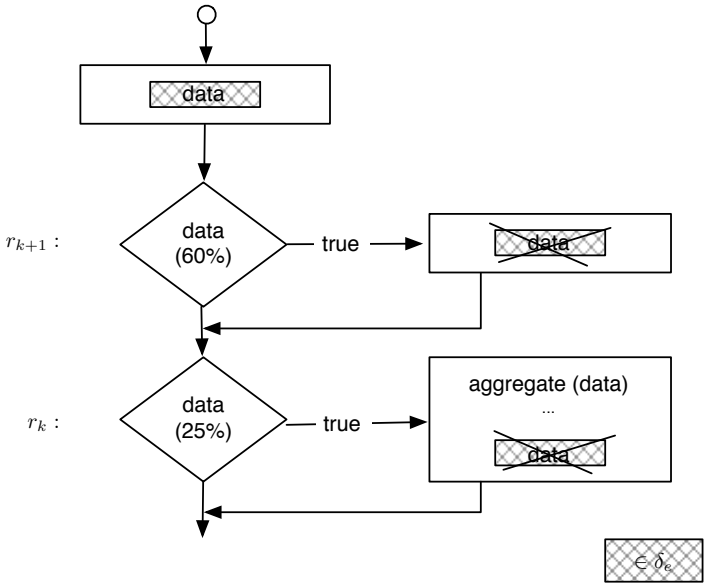


Figure 5.4: Chaining of filters for efficient data processing

rules have only to be evaluated for a fraction of all incoming facts since the early eviction of the initial trigger renders condition evaluation superfluous. Rules such as e.g. rule r_{k+1} that filter 60% of all facts named data are quite common in wireless sensor network applications: Usually, only a small value range for data samples is of interest. Outliers can then be immediately dropped and remaining data items instantly aggregated (see rule r_k). Note that it is however essential to retract or modify the incoming fact in the body of a processing rule to either remove it from the fact repository or swap it to the set of modifications δ_m to achieve the effect of successful evaluation suppression.

The positive effect of incorporating this pattern into the data processing chain is twofold: On the one hand, the search space for pattern matching during condition evaluation is kept small if only significant data is kept. Furthermore, the utilization of the fact repository is optimized. On the other hand, the unnecessary evaluation of rules is suppressed in case triggering facts are evicted early. Naturally, this can, dependent on the number of rules, speed up the rule evaluation process tremendously and, due to a lesser workload on the CPU, save valuable energy otherwise wasted for needless processing.

Hierarchical Data Fusion for Complex Event Recognition

While wireless sensor nodes are capable enough to execute basic data processing and compression algorithms, storage for data samples is often a bottleneck. Dependent on the deployed platform and the application itself, two general ap-

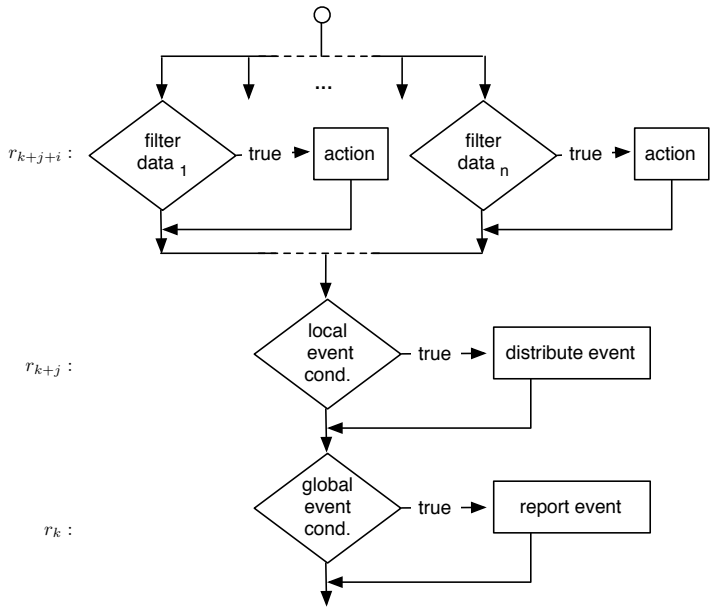


Figure 5.5: Filtering and fusing data for complex event detection

proaches how to deal with these samples can be observed. Monitoring applications that focus on pure data acquirement predominately build upon sensor network hardware with large, secondary storage. During the deployment of the network, samples are swapped to storage and information passing in-between nodes is kept to a minimum. A low energy consumption due to communication avoidance and the availability of all raw data at the end of a deployment are major advantages. Then again, network behavior is rather passive, event detection can only be performed offline and clearly the great potential of the sensor network technology is not exploited.

Whenever sensor nodes are deployed to immediately react to recognized events, the data management strategy is naturally different. In this case, all relevant data has to be accessible fast so that excessive swapping and searching can be prevented. Instead, a pattern implementing a hierarchical data fusion algorithm, see Figure 5.5, is often used to cut data size, yet maintain crucial information and at the end enable the inference of complex or composite events [31]. Therefore, a concatenation of several processing steps that can be nicely mapped to rules is necessary, typically starting with adequate filtering as described in the pattern presented above. In a next step, aggregation schemes which operate upon samples of the same kind and/or data fusion algorithms [40] to relate different types of aggregates are applied. The goal is to semantically compress raw samples to local events, thus add meaning to the data whilst optimizing for storage at the same time. Numerous publications have explored this topic, e.g. [33], [58], its

corresponding implementation in rules is straight-forward.

As soon as local events have successfully been recognized on an individual node, the application may have instructed it to react in a variety of ways: Actors available on the sensor node may be activated, information may be passed to a dedicated source across the network or further in-network processing may be triggered, as depicted in rules r_{k+j} and r_k in Figure 5.5. Here, the detection of an event is verified by means of querying the neighborhood for their results, achieved by distributing the local event. In case the conditions for global event detection are met, which can for instance be implemented as a simple majority vote, the sensor node that initially requested reassurance will fire an appropriate reaction.

It is noteworthy to point out that the sheer size of data which has to be stored on a node can be tremendously minimized when utilizing this pattern. Each step up the hierarchy increases the level of compression, moving from raw data samples over aggregates to local events, which may then even be fused to events that represent a region covered by a set of sensor nodes. Not only is this beneficial regarding node-local storage capacity, but it also helps to keep network load low as lesser traffic is imposed when relying on compressed data to forward. For concrete numbers on possible savings and a detailed implementation of this pattern in RDL rules, the interested reader may be referred to an experiment described in [154]. Also note that aggregation schemes may well be applied whilst routing events from sources to a data sink, see e.g. [147], which clearly is just a different approach towards implementing this pattern.

5.3.3 Remarks

Real-world applications and protocols are often composed of autonomous processing parts or phases which are plugged together to achieve the envisioned behavior of a node. From a software engineering point of view, it is beneficial to map such parts to well-known programming patterns where possible. This not only clarifies needed control and data structures, but also adds to the readability of the implementation and may speed up the development process.

The patterns presented above have been retrieved from different rulesets and implementations that were developed with RDL. For successive processing of facts of the same type that need to implement many-to-many matching semantics under state-based constraints, the utilization of the `while`-loop has for instance been very helpful. This has been used to find adequate routing options in the PST routing example which will be discussed in detail in Section 7.2. Complex event detection, thus the definition of causal and/or temporal dependencies between sensor samples, derived events and e.g. network state, can in most cases be easily mapped to a state machine. Reaching a specific state presumes a primitive event to have occurred, and rules for fusing state information can act upon these state facts to recognize the incidence of a complex event scheme. A representative implementation is for instance the cow tracking example presented in the motivation to this thesis. Finally, fact casting for generic matching has been a widely applied

pattern. A very basic task that can be nicely implemented with it is for instance the definition of a holistic aggregation scheme for sensor samples. For long term environmental monitoring, the individual samples are usually of minor importance and it is enough to provide solely the value distribution by extracting average and median values for a specific sample period and all available sensors. Since sensor samples usually resemble the same fact structure, the pattern allows to specify aggregation functionality only once, and invoke it on the different samples.

Both, the *chain of filters* and the *hierarchical data fusion* pattern basically stem from the same, real-world experiment which is discussed in detail in Section 7.4. There, the number of samples streaming into the system have been almost overwhelming in case a local event triggered, so that fast filtering was mandatory to actually keep the system up and running. The more semantics were added to the raw data, the less the sheer volume became, a circumstance found very helpful for speeding up the evaluation process along the line of complex event processing schemes.

In general, these patterns are in no way comprehensive, which was however never the motivation. Rather than elaborating on all possible and useful patterns, the prime intention has been to provide some hints on how to implement recurring algorithmic sequences that are *not* directly covered by language constructs, and how to cope with common WSN specific challenges.

5.4 Towards mature language design

The RDL language presented so far is absolutely suitable to implement a wide range of wireless sensor network protocols and applications. Stand-alone, all relevant processing and communication capabilities are provided to efficiently denote reactive, node-local behavior. However, to serve as a language that enables a more mature approach towards software development, thus exhibits constructs as e.g. support for modularity that programmers are nowadays used to from high-level programming languages, relevant features are missing. It is also unclear, whether the complete lack of imperative control sequences has been a good choice, or whether an integrative approach would be preferable

In this section, two approaches to enhance or to alter the core language by provision of additional functionality are briefly introduced and discussed. The first aims at studying a means to integrate the imperative and the rule-based paradigm at the language level. The question has been whether there is a possibility to take the look and feel of a more familiar programming language such as for instance C# and integrate an RDL derivative that basically promotes the same functionality. It turned out to be a quite experimental approach resulting in a conceptual language called *Small#* with very interesting features, which in the end however lacked the capabilities of the core language.

The starting point for the second study has been the unchanged, core language itself. Here, the focus has been to provide additional, lightweight language con-

structs that enable a better software design in terms of established, compile-time support for increased code stability and better encapsulation of concerns [84].

5.4.1 Small# - A conceptual approach towards paradigm fusion

Unlike the development process for RDL where an analysis of demands and goals from a problem-oriented point of view led to the final language syntax and semantics, the design process pursued for Small# has been bottom-up. Available hardware platform, system interface and the denoted requirement to integrate RDL rule specification semantics into a stripped-down C# for embedded devices were taken to weave a programming fabric. Design thus started with crafting a suitable instruction set in combination with a specification of supported data types. Based upon this set of available instructions, a high-level representation depending on a familiar syntax has been developed to enable sensor node tasking. Once again, an interpreter architecture was chosen to warrant correct execution semantics. Note that a complete reference of the final instruction set architecture, the implementation of a virtual machine (VM) and the Small# syntax can be found in [117]; in the following, the goal is solely to sketch the initial idea and discuss its practicality.

The Small# virtual machine instruction set architecture (ISA)

Bottom-up design usually starts from picking up the pieces that are already at hand, a fact that is confirmed in the Small# ISA. The definition of data types is straight-forward, exporting common types such as boolean, unsigned integer and so forth directly from the underlying system to the ISA. In addition, the data types `Function, Address, Slot` and `Fact` are given, with the first three being references to memory and the last being an index to a separately managed memory region for fact storage.

The ISA targets a RISC-inspired load/store architecture, however with variable-sized opcodes to prevent memory wastage later on in the compiled bytecode. The actual instruction set comprises 60 instructions, which can be categorized into six different groups:

- Instructions that provide basic load/store operations for stack-based program execution
- Instructions to manipulate program flow
- Instructions for arithmetic and logic operations
- Instructions for direct fact manipulation
- Instructions that allow for controlling communication devices (radio, serial interface)

- Instructions that directly export offered firmware functionality to manipulate and control the sensor hardware

Each instruction is encoded with a distinct opcode and an operand if necessary to obtain a concise bytecode, which can then be interpreted by a Small# virtual machine respectively. What is now interesting to see is how rules, facts and slots are represented at the instruction set level and intended to be integrated within the virtual machine. This will later on determine the access strategy that is available from a language that compiles down to the given bytecode format.

A function comprises a sequence of instructions that semantically belong together. Representing a function via an address that indicates the starting point of this sequence within the bytecode is therefore a natural approach. A rule can be understood as a set of two consecutive functions, one that evaluates the rule precondition and one to be eventually called via a jump instruction at the end of successful condition evaluation implementing the required actions. Finally, also a slot can be perceived as a mere sequence of filtering conditions, yielding a bytecode representation of both, a rule and a slot, by provision of the address of the sequence starting point.

Facts are a specialized format to encode data of global scope. Since they are not automatically managed, they require proper handling by a virtual machine at runtime. Dependent on the VM implementation, an address that points to a reserved memory region for that fact, or an index to an element of a data structure chosen for storage as suggested in this ISA can be utilized for fact encoding.

As can be seen from the ISA layout, a stack-based instruction execution model is the basis for the architecture: all fact processing, reactions and filtering mechanisms are denoted as functions. Parameters such as variables, constants or fact addresses are pushed onto the stack and evaluated according to the decoded instructions. This design choice clearly mirrors the intent to integrate rule-based processing capabilities into imperative programming - and not vice versa.

The Small# language

While it is possible to write assembler-lookalike programs with the ISA briefly introduced, a decreased level of abstraction will most certainly not satisfy future developers. The provision of a high-level language that can compile down to this intermediate bytecode is a more rewarding approach. Listing 5.1 visualizes an example that implements periodic temperature sampling in Small#, a potential language syntax for a hybrid C#/RDL approach.

Listing 5.1: Simple temperature monitoring in Small#.

```
1  class MonitorTemperature {
2
3    const int maxTemp = 25;
4    int sum, count, average;
5
6    //filter requirement for a temperature fact
7    slot temperature = Temperature (this.state == "modified" , this
         .time < System.getCurrentTime());
8
9    //Function to trigger an alarm
10   void TemperatureExceeded(int current) {
11     System.SerialWriteLine("Current" + current + "Average" +
           average);
12   }
13
14   //rule that fires upon new temperature facts
15   rule TemperatureMeasured (100)
16   cond {System.Exists(temperature)}
17   inst {
18     sum += temperature.Value;
19     count += 1;
20     average = sum/count;
21     if (temperature.Value > 25) {
22       TemperatureExceeded(temperature.Value);
23     }
24     else {
25       System.SerialWriteLine("OK");
26     }
27     System.DeleteFact(temperature);
28   }
29   //Entry point, initialize temperature sampling
30   void main() {
31     sum = count = average = 0;
32     System.SampleTemperature((60 * 1024), 0, 0);
33   }
34 }
```

Similar to C# programs, the `main` function is the entry point to start the execution of the *MonitorTemperature* program. Here, the global variables *sum, count* and *average* are initialized to zero, and a function that is defined in a separate *System* class to trigger periodic sampling of a temperature sensor once a minute is called. Program execution then stalls, and will only be re-activated upon the occurrence of a fact named *Temperature*. This matches the name of the fact denoted in the condition specified in line 16, which in turn points to slot *temperature* (lines 7 - 8) for predicate evaluation. The rule *TemperatureMeasured* will trigger in case the state property of a *temperature* fact is set to modified and its creation time does not lie in the future, a simple sanity check. Then, the action part of the rule (lines 18-28) is executed, the average temperature calculated, and

an alarm message printed to a serial interface in case a predefined threshold value for the current measurement is exceeded. At the end of the processing cycle, the fact referenced by the temperature slot is retracted from the system.

Note that while the bytecode has been extensively tested on a dedicated virtual machine for the target ISA, the language itself has never passed the conceptual stage. The reasons for this will be reconsidered in the subsequent section.

Discussion

The central question that needs to be answered to judge a hybrid approach as for instance the Small# language is what the actual benefit in contrast to either a pure rule-based or a pure imperative language for WSN tasking is. As can be directly drawn from the design of the ISA and the presented sample code, the prevalent view on software development has changed from a problem-oriented (RDL) to an instruction-oriented perspective which is typical for the imperative paradigm.

The provision of local variables, the ability to specify functions that can be called from the rule body and the availability of instructions to define control sequences for sure facilitate the implementation of sophisticated data processing schemes: In contrast to RDL, values of matched fact properties can be bound to variables and common sequences of operations are directly accessible from the action part of a rule. This can be done by calling a subroutine instead of wrapping the behavior into another rule. Even *if-then-else* statements can be part of an action and do not have to be outsourced, which most probably increases code readability.

But all the advantages named above could have been equally achieved when relying on the implementation of simple event handlers as presented in Section 2.2.3. In this case, an interpreter architecture would be obsolete, granting increased processing speed at lesser sourcecode to maintain. However, some features that are gained when adding the fact-based data model and the rule abstraction to imperative programming are then lost. Recall that event handlers follow a simple one-to-one matching scheme, thus the occurrence of an event will invoke exactly one handler. The ability that distinct processing paths can be followed at the same time as provided by one-to-many matching semantics is not supported. Also, a unified data model as enforced in RDL and at least supported by Small# substantially abstracts from machine-driven data handling, hence allows for a more problem-oriented utilization of data items.

Nevertheless, the hybrid programming model introduces some inherent, major drawbacks that rendered further research in this direction rather fruitless. First and foremost, the availability of local variables and primitive data types in combination with the fact model breaks the introduced data and processing abstraction with severe consequences: As a stack-based processing approach is chosen for execution and a developer is able to manipulate this stack according to his needs, manual stack management is pushed back to be his responsibility. At the same time, the availability of language constructs for explicit flow control allows for the

definition of long-running functions, possibly leading to timing problems when utilized on embedded hardware. In contrast to pure RDL execution schemes where processing concerns are completely shielded from the developer and handled in the runtime environment, this is not feasible any more within the hybrid model. Second, redundancy is introduced into the programming model which is of course a result of paradigm fusion: One can for instance write a function that evaluates the rule conditions by means of iterating over the fact repository, or one could rely on the functionality provided by a runtime, which will trigger the same action when questioned to evaluate a slot. Also, *if-then-else* sequences can be applied to avoid rule specification, which is the case in the example program, or, the same processing semantics may be achieved by specification of additional rules. Of course, a possible workaround is to get rid of one or the other, but then either paradigm is marginalized.

Summing up these observations, the disadvantages that the integration of paradigms at this level and for the given purpose introduce, clearly outweigh the advantages gained. Despite of this rather disappointing result, the evaluation of the Small# approach however pointed out important features of the pure programming paradigms that otherwise might have not been revealed in this clarity.

5.4.2 mRDL - Enhancing RDL for improved modularity

The Small# approach focussed on exploring whether an adaptation of RDL by addition of imperative control and data manipulation constructs is plausible. In contrast to this, the approach presented and discussed in the following is concerned with staying within the conceptual bounds of the language and analyze, once again with a top-down approach, whether RDL misses important features, what exactly their nature is and how these shortcomings can be objected.

A first hint towards a missing language feature has already been mentioned in Section 5.3.1 where the need for a mechanism to reduce code obfuscation has been tackled. Looking back, one can read between the lines, that this is not only an issue at the granularity of rules, but can be extrapolated to complete rulesets. Within a ruleset, functional polymorphism can be achieved with a guarding rule that exports a specific fact as an interface, but how to cope with the problem at a larger scope has not yet been discussed. In general, the open question is how to specify a reasonable encapsulation of concerns with the ability to share and to make the provided implementation accessible in a coordinated, structured manner. Note however that this is not a question of a functional deficiency of the language, but rather a question of missing language support for improved software engineering techniques.

Missing pieces for support of modular software development with RDL

In general, modular software development is concerned with encapsulating software parts that functionally belong together into a so called software component.

Each component consists of the implementation of a requested operation in combination with the therefore required data structures, and an interface which exports a handle to utilize this functionality [110]. Earlier we discussed that due to a lack of sequential program flow for custom rule-based processing, it is not reasonable to integrate language constructs equivalent to subroutines into a rule-based language. At a larger scope however, thus when it comes to the implementation of either complete protocols or functionally independent sets of control rules, the option for code reuse becomes an interesting feature. In RDL, the equivalent to a software component can be denoted to be a ruleset.

Since the mere existence of a keyword to specify software components is not enough, a number of issues still have to be addressed to be able to actually depend on sharing ruleset functionality:

- *Ruleset interdependency:* While it is possible to assign specific rules to belong to a certain ruleset, RDL features no mechanism to denote the correlation of different rulesets.

- *Namespaces:* Facts are global variables, hence have global scope and names are visible throughout the rule base. In case several, independently developed rulesets are used together, erroneous node behavior due to a clash of names is inevitable with core RDL syntax.

- *Information hiding:* Unless explicit access restrictions are introduced, rules from different rulesets can interact, thus trigger each other, whether this is intended or not. Encapsulation of concerns cannot be granted with core RDL.

Syntactical changes in mRDL

A couple of mechanisms have to be added to address the shortcomings pointed out above and embed countermeasures into the language. First of all, it is necessary to provide a means for specification of ruleset interdependency. For instance, if *ruleset A* comprises rules that express the application logic, and *ruleset B* rules that implement a data streaming protocol, then for A to use B, a mechanism has to be available to instruct the compiler that (1) A is not operational without B, thus A depends on B, and (2) in which order rules of these rulesets should be checked. Similar to an `import` instruction for classes in Java or an `include` instruction in C to point to relevant header files, the RDL syntax is therefore enhanced by the keyword `depends`, indicating that a ruleset cannot be utilized without the availability of the ruleset that it declares to depend on. Furthermore, since scheduling semantics within a ruleset should be kept, evaluation order amongst rulesets has to be denoted with an additional assignment of priority. Typically, the lower in the stack implemented functionality can be categorized, the higher is the assigned scheduling priority.

Since facts serve as a means of internal communication of system state between rules for forward chaining, the probability of an unintentional clash of names when sharing a global namespace among rulesets increases with the number of rules provided. A simple, yet effective mechanism to avoid this, is to enable the declaration of namespaces, which restrict the scope of names to the rulesets they have been declared in. Unless explicitly addressed, fact names outside the current ruleset are then not visible. As a consequence, the danger of unintentional trigger provision during ruleset specification is avoided.

Nevertheless, without a means to support ruleset interaction, the concept of modular software development is rendered useless. The key to controlled rule 'invocation' is clearly to provide (1) a dedicated interface that exports the fact signature, thus its internal structure to the issuing ruleset and (2) a mechanism to allow for cross-ruleset visibility and addressing of fact names. The straightforward approach to embed these requests into RDL is to force a programmer to explicitly declare any name visible across ruleset boundaries and to set it to be `public`. The interface to a ruleset therefore constitutes itself in a list of names of corresponding facts that will trigger rules when inserted into the fact repository. In order to utilize and address a fact of a foreign namespace, the rulesets' name followed by a period has to precede the factname in question.

Given a ruleset A which wants to trigger a rule r_b in ruleset B, one of the triggering facts for this rule has to be declared public. Once again, the issue of generic matching becomes apparent in case the fact is supposed to serve as more than a mere trigger, but will be altered or used for parameterization of the rule intended to trigger. In the present model, only a mechanism that resembles 'call by value' semantics is supported[3], a circumstance that, in respect to the predominantly utilized pattern matching, is not optimal. We therefore experimented with introducing the possibility of passing references to fact types, and apply matching upon reference resolution. However, in practice this approach turned out to be very prone to errors due to utterly complex slot specification schemes and resulted in ambiguous semantics for matching fact properties on referenced fact types.

Practical relevance of mRDL

Support for modularity is not only beneficial from a software engineering point of view, but is especially in the context of wireless sensor networks of practical relevance for differential updates. Recall that a prime application area of WSNs is their utilization for environmental or habit monitoring. Software updates necessary during the time of deployment are resource-intense operations as the image has to be diffused to all target nodes over-the-air, which is costly in terms of energy spent. Furthermore, the image has to be stored in memory until all parts fragmented over a set of update packets have been received, taking up valuable

[3]Be aware that this is a leaky metaphor since rule-based processing depends on matching, not on method invocation.

memory capacity. The smaller the update, the faster is a node able to resume its actual task and the less energy is wasted.

Differential updates at system runtime are especially easy to support when the deployed software is already structured in a modular manner. Since each ruleset hides its implementation behind its interface, their exchange is unpretentious as long as the interface is kept regardless of the severity of change. A tremendous number of protocols for differential updates of native images [101], bytecode [93] and even complete reflashing [82] at runtime have been proposed that address issues such as efficient image distribution [71], state management [137] and network synchronization [94]. Further research targeting this direction has therefore not been undertaken within this thesis.

Outlook and Discussion

One feature that has not been addressed in the above mentioned enhancements is the type system. RDL supports only a small subset of available data types that are commonly utilized in full-fledged programming languages. For all of the experiments carried out so far, this has been a sufficient choice. However, in case RDL is utilized for the implementation of more sophisticated problems that e.g. demand for high-precision data samples, the available set naturally has to be adjusted. Also, for better error checking capabilities at compile time, the introduction of a strict type system which is currently not enforced in RDL is a reasonable refinement.

5.5 Concluding Remarks

In this chapter the domain-specific, rule-based programming language RDL has been presented in syntax, semantics and pragmatics and discussed in depth. The extent of the language syntax itself is rather small and features only a limited set of simple grammar elements: rules, facts and slots, binary and unary expressions, basic fact manipulation schemes and an interface to the underlying hardware. Nevertheless, their application in a nested manner is especially powerful to specify complex filtering predicates operating on the fact repository, adding to the language's expressiveness.

The denotational semantics revealed interesting insights on actual language usability and clarified how exactly RDL rules have to be conceived: Reactivity is the foremost concern that the language absorbs, however not in a manner congruent to the ECA paradigm. Instead, events are persistent and map current system state within the fact repository. Furthermore, the evaluation process of rules follows stateful semantics, meaning that changes introduced by a rule are directly accessible by subsequent rules. At the same time, correct event and rule evaluation order are strictly enforced as a facts' ability to trigger rules is delayed to emerge at the beginning of a new production run. As a result, production rule semantics *and* sound event processing can be supported at the same time.

In case intended application or protocol semantics do not follow an event-driven or knowledge representation approach, but rather build upon typical imperative processing schemes, the utilization of RDL is not optimal. Due to a lack of directly accessible mechanisms to control program flow, a developer has to exploit language evaluation semantics to obtain the requested processing sequence. To alleviate this task, several programming patterns derived from running RDL programs have been presented and discussed. With these at hand, unorthodox language utilization is facilitated, opening up a wider spectrum of possible application domains for RDL.

Since the request for support of common imperative programming sequences became apparent quite early during the language development process, an approach to fuse language constructs from both, the rule-based and the imperative paradigm, has been explored. Already observable at the level of the instruction set, but clearly visible within evaluated Small# sample code, this approach however turned out to be not feasible: excessive redundancy, a lack of a clear data model and no real advantage over any of the pure approaches resulted in project termination. Another approach that has however been successfully tested was to amend the core language RDL to support a modular software development process. The addition of a few, simple grammar elements has had the desired effect so that related practical issues such as differential reprogramming capabilities may now easily be integrated into the framework.

All in all, this chapter demonstrated RDL to be a language that provides the designated programming abstraction. It turned out that even when facing suboptimal envisioned program structure, RDL is nevertheless a valid and operational choice, however at the cost of explicit exploitation of language semantics.

Chapter 6

Implementing FACTS

For actually running rules on sensor nodes or within network simulations, these have to be made available, manageable and interpretable. As has been highlighted in Section 4.1 for reasons of increased portability, supervised stack utilization and better retasking capabilities at runtime, the chosen model to deal with user-specified rules is that of a runtime environment which interprets rules compiled to a concise bytecode. The focus of this chapter is on the implementation of the necessary components to enable rule interpretation, as well as on the different parts of the FACTS toolchain.

Based on the fact that subsequent sections build upon the results of the compilation process, this chapter starts off with a detailed description of the steps undertaken for bytecode generation in Section 6.1. With the help of well-known tools for lexical analysis and parsing, the RDL compiler, often also labeled as *FACTS-rc*, has been implemented to create a viable bytecode for later on interpretation. Several optimizations to gain a leaner bytecode and to speed up bytecode evaluation at runtime have been explored and will therefore also be part of the discussion in this section.

RDL rules can not only be executed upon the ScatterWeb MSB430 sensor nodes, but compilation targets also include one of the standard network simulators, `ns-2`, and a Haskell backend which has primarily been used during language specification. While the emphasis of Section 6.2 is clearly on practical aspects of rule engine implementation, it will as well introduce the different backends available for RDL interpretation and reveal their general design.

This chapter is finalized by a discussion of issues that have to be taken into account for designing a sound, easy-to-use interface to sensor node hardware in Section 6.3 given the perspective of an application developer. The need for ultimate control of sensing components to achieve optimal application configuration and the equally important support for rapid prototyping have to be balanced to provide a platform that is suitable throughout application development time. This request is objected by two mechanisms presented in this section, before Section 6.6 summarizes the findings of this chapter.

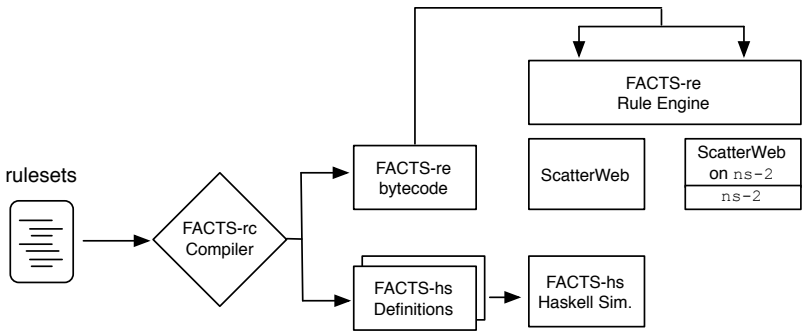


Figure 6.1: Compilation targets and bytecode generation for FACTS

6.1 Rule Compiler

Rules denoted in the RDL language have to be compiled to bytecode in order to enable their execution in different runtime environments. Within the FACTS middleware framework, the ruleset compiler (FACTS-rc) therefore implements the necessary steps to parse input rulesets into an abstract syntax tree (AST) and generate interpretable bytecode from the obtained data structure. Figure 6.1 depicts the different backends that may be target to the compilation process to provide an overview of involved components, which will however be revisited in the following section.

6.1.1 Basic Compilation

The first step towards compilation is the lexical analysis of a given program. The starting point for this process is the lexical grammar of RDL as provided in Section 5.1.1, which allows the corresponding *lexer* to form tokens from the input stream of ASCII characters. These are afterwards passed to a parser for syntactical analysis. To this end, we depend on the freely available tools `lex` and `yacc` [3] for convenient lexer generation for RDL and parsing. During these steps, the input program is checked whether it conforms to the defined syntactical structure a ruleset has to exhibit and will be allocated into the data structure of an abstract syntax tree accordingly. Otherwise, an error is thrown indicating the line that troubles the lexer or parser.

The interesting part is now how to proceed with the obtained AST, or more precisely, how to structure the bytecode to gain a well-formed and concise layout of the envisioned interpretable source. This is especially interesting for the bytecode targeting the ScatterWeb backend, since in this case, execution time is an important design rationale. Generation for the Haskell backend is a straight-forward, syntactical transformation of the AST into Haskell definitions, thus does not reveal any new aspects beyond those discussed for FACTS-re bytecode generation,

and will therefore be left out here for reasons of brevity.

In general, two different approaches are available to encode an AST, both with clear advantages and disadvantage.

- *Linked Lists*: The obvious solution to transform the AST into interpretable code is to utilize linked lists for storing the relationship between rule parts. For each non-terminal element, a list of pointers to its parts has to be created and, during the interpretation process, resolved, an approach that preserves the initial AST as well as data locality. However, the overhead for using pointers grows in a linear manner with the number of elements in a list, a circumstance that has a direct, negative effect on bytecode size.

- *Address-based Encoding*: Another option for encoding is to resort to an address-based scheme in combination with distinct arrays for individual element types. The basic idea here is to count all rules, conditions, statements, facts and so forth, allocate an array for each type respectively, and encode the relationship between these arrays. A rule, which is a mere sequence of conditions and statements, will hence be encoded by denoting the start and end address of its conditions in the condition array, and of its statements in the statements array correspondingly. While locality of data is lost due to sorting the tokens by type and writing them into an array in a sequential manner, the advantage is clearly the constant encoding overhead of maximum two addresses per element, thus independence of ruleset structure.

Since bytecode conciseness has been one of the design rationales, the second approach has been chosen for the FACTS-rc compilation process, with several modifications being additionally integrated to further compress the compiler output. With the AST as input to the compilation process, FACTS-rc starts from the items that are first accessed during rule evaluation, thus rules, to lay out the bytecode for its easy traversal at evalation time. All tokens that are part of the blocks, i. e. named slots or names, are directly copied to the location in the data structure where they are actually referenced. Naturally, this will increase bytecode size at first, since code can well be duplicated, a circumstance however tolerated due to facilitated optimization which is later tackled in the compilation process.

Each parsed item is then either encoded by the addresses into the corresponding arrays of its terminal and non-terminal items it is composed of, or, in case a sequence of items is referenced, by their start and end address. An alternative solution would have been to denote the start address and a value representing the number of items in the sequence of interest. However, this would have resulted in the allocation of an additional automatic variable per sequence for counting during rule evaluation. Due to deep nesting of such sequences, the incurred stack overhead can become significant, so that this choice has been discarded.

The encoding scheme for strings follows the request for small bytecode sizes: No string is preserved in the bytecode, but mapped to a unique integer during

Figure 6.2: Abstract layout of a FACTS bytecode image

compilation instead. If necessary, the separation between rulesets is kept by retaining the namespace scope of names of facts, slots and names. This (lossy) compression method is valid since no direct user interaction at runtime is necessary. In order to ease debugging, the compiler will create a separate file which denotes these key/value mappings along with the bytecode and can be consulted if facts are printed over the serial interface.

Figure 6.2 shows the abstract layout of the compact structure of a FACTS-re bytecode image after compilation. All tokens are simply parsed into their individual arrays and then combined into one image to be interpreted on the sensor node.

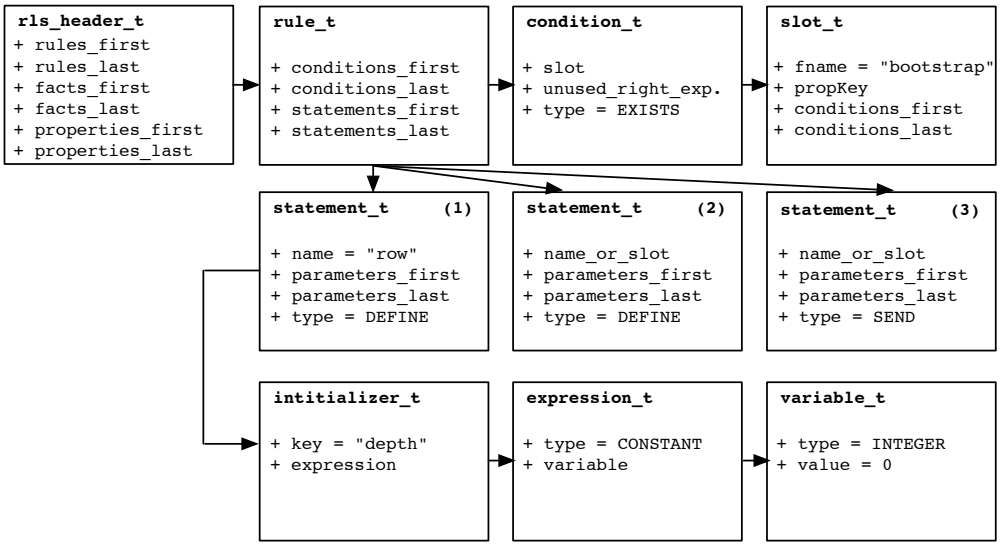
To summarize the steps discussed above, Figure 6.3 illustrates this complete process from rule definition to image layout, skipping however the AST generation for conciseness. An in-depth discussion, along with a complete example, can be found in [153]. In the first part, one out of the rules of the *"Xmas"* ruleset is given, see also Appendix B.1 for a complete implementation. This rule consists of one condition and three statements to be executed upon event recognition. The second part gives an idea of the data structures a ruleset is parsed to for further processing; it declares a ruleset to consists of rules, facts and properties, a rule to be denoted by its conditions and its statements and so forth. Finally, the actual layout of the bytecode is depicted in the third portion of the figure, visualized in its hexadecimal representation for simplified reading. Note that the last part is actually intended to be interpreted by the ScatterWeb nodes and not read by human beings and none of the parts is a complete representation of the chosen ruleset.

Each rule image starts with a magic number, the sequence of hexadecimal numbers `0xC7` and `0xFA` (first row, first column at addresses 0x1000 and 0x1001 in the image) which have to be available at the beginning for the rule engine to recognize the bytecode to be valid. Following this, the header information of the

Rule Excerpt

```
[...]
rule getNumRows 100
<- exists {bootstrap}
-> define row [depth = 0]
-> define light [on = true, node = 0]
-> send systemBroadcast systemTxRange {row}
[...]
```

Bytecode Structure



Bytecode Layout

```
           00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
1000:100F  C7 FA 14 10 74 10 70 16 70 16 84 16 8A 16 00 00  ┐ 0x1002-0x1013: header
   ...     00 00 00 00 80 10 80 10 04 11 14 11 00 00 00 00  ┘
           86 10 86 10 1C 11 34 11 00 00 00 00 92 10 9E 10  ┐
           3C 11 4C 11 00 00 00 00 A4 10 AA 10 54 11 6C 11  │
           00 00 00 00 B0 10 BC 10 74 11 8C 11 00 00 00 00  │ 0x1014-0x107F: rules
           C2 10 C8 10 94 11 BC 11 00 00 00 00 CE 10 D4 10  │
           C4 11 DC 11 00 00 00 00 DA 10 E0 10 E4 11 EC 11  │
           00 00 00 00 F2 10 F8 10 F4 11 FC 11 00 00 00 00  ┘
1080:108F  04 12 00 00 00 00 24 12 00 00 00 00 22 14 28 14  ┐
           0D 00 64 12 00 00 00 00 46 14 4C 14 0D 00 52 14  │
           58 14 01 00 84 12 00 00 00 00 70 14 76 14 01 00  │
           B4 12 00 00 00 00 9A 14 A0 14 05 00 A6 14 AC 14  │ 0x1080-0x1103: conditions
           09 00 14 13 00 00 00 00 D6 14 DC 14 15 00 64 13  │
           00 00 00 00 06 15 0C 15 05 00 9C 13 00 00 00 00  │
           24 15 2A 15 01 00 30 15 36 15 01 00 3C 15 42 15  │
           05 00 CC 13 00 00 00 00 4E 15 54 15 01 00 5A 15  │
           60 15 05 00 0A 00 6C 15 6C 15 00 00 0C 00 70 15  ┘
1100:110F  74 15 00 00 0C 12 16 14 1C 14 03 00 3C 12 2E 14  ┐
           00 00 04 00 4C 12 00 00 00 00 01 00 13 00 34 14  │
           34 14 06 00 15 00 3A 14 40 14 06 00 16 00 78 15  │
           7C 15 00 00 19 00 80 15 80 15 00 00 7C 12 00 00  │
           00 00 01 00 94 12 7C 14 00 00 04 00 1C 00 8E 14  │
           8E 14 06 00 A4 12 94 14 00 00 04 00 AC 12 00 00  │
           00 00 01 00 D4 12 B2 14 00 00 04 00 E4 12 C4 14  │
           00 00 04 00 F4 12 CA 14 D0 14 03 00 0C 13 00 00  │ 0x1104-0x1203: statements
           00 00 01 00 2C 13 E2 14 00 00 04 00 34 13 E8 14  │
           00 00 04 00 1C 00 EE 14 EE 14 06 00 3C 13 F4 14  │
           FA 14 03 00 54 13 00 15 00 00 04 00 5C 13 00 00  │
           00 00 01 00 12 00 84 15 84 15 00 00 7C 13 18 15  │
           1E 15 03 00 8C 13 00 00 00 00 01 00 94 13 00 00  │
           00 00 01 00 1C 00 48 15 48 15 06 00 C4 13 00 00  │
           00 00 01 00 1C 00 66 15 66 15 06 00 E4 13 00 00  │
           00 00 01 00 0C 00 00 00 00 00 00 00 22 00 00 00  ┘
1210:121F  00 00 00 00 0F 00 10 00 00 00 00 00 0F 00 11 00  ⋮ 0x1203 - 1423: slots
```

Figure 6.3: Transformation of the ChristmasLights ruleset to bytecode

ruleset, thus the starting and ending addresses of rules, facts and properties are denoted in the bytecode. If kept in mind that addresses are read backwards, one can easily see in this image representation that rules are encoded starting from address `0x1014` to address `0x107F`, with the last rule starting at `0x1074`.

In case we jump directly to address `0x1014`, highlighted in yellow, two subsequent numbers that point to address `0x1080` can be found. Since the first rule of the ruleset, the rule named *getNumRows* has only one condition, both the start and the end address for the conditions of this rule are the same. The three statements of rule *getNumRows*, are encoded from address `0x1104` to `0x1114`, and highlighted in green respectively. Before the next rule is encoded, the two following zeros are due to an enhancement further described in Section 6.1.3. As can be derived from the bytecode, the complete ruleset consists of nine rules, encoded from address 0x1014 to 0x107F,

Starting from address `0x1080`, the condition of rule *getNumRows* is encoded. In general, a condition is composed of a slot (or a left expression), an unused value (or a right expression) and the type of the condition, which can be either of type `EXISTS` or of type `EVAL`. Since in this case, an `EXISTS` condition is given, no left and right expression are referenced, indicated by the zero values, whereas the slot is available starting at address `0x1204`, which is once again highlighted, this time in magenta. Here, only the name of the slot, the value `0x000C`, i.e. the integer value that the string "bootstrap" has been mapped to during compilation, is non-zero. No conditions and no property key have to be encoded for this slot.

The first statement out of the three can partially be followed within the bytecode image at the bottom of the figure. The encoding provides an address for a slot named "row" at address `0x120C`, thus right after the magenta-coloured part of the image, as well as its parameters ranging from address `0x1416` to address `0x141C` and the type being a `DEFINE` statement. Parameters are however not displayed in the image, but solely exemplarily represented in the middle part of the figure. The string "row" is in this ruleset mapped to the integer value `0x0022`.

Overall, the complete *"Xmas"* ruleset is compiled to 1676 byte in this unoptimized manner. In the next section, we address this rather big bytecode size by introducing a way to optimize the compilation process. In order to give a first impression on average ruleset sizes and statistics in terms of language constructs, Table 6.1 depicts some numbers from different ruleset that will be further discussed in the next section. From a first glance, one can see that the mere number of rules per ruleset can range from only four, up to more than fifty dependent on the algorithmic complexity of envisioned node behavior.

6.1.2 Bytecode Optimization I: Smaller Image Size

In contrast to image generation for imperative programs, the layout of a rule-based bytecode can be composed and optimized with almost no restrictions. Solely two requirements have to be met: On the one hand, the order of rule evaluation has to be preserved at execution time according to the ordering specified by the

Table 6.1: Statistics of selected rulesets

	Xmas	Turing	DD	GRA	FROMS
Rules	9	4	18	14	51
Conditions	22	7	46	63	331
Statements	32	10	55	35	205
Slots	20	21	34	64	829
Expressions	20	20	35	61	782

programmer. To prevent explicit encoding of rule priorities with each rule and to bypass costly search operations within the bytecode for the next rule to check at runtime, the ruleset is sorted in descending order prior to compilation so that the rule engine may resort to sequential analysis. On the other hand, sequences of statements have to be maintained to achieve correct application behavior and sequences of conditions cannot be broken into parts to not obscure the encoding scheme based on start and end addresses. Other than this, one is free to organize and re-structure the image layout to gain leaner bytecode.

As has been pointed out, the basic encoding scheme presented in the previous section still features a good deal of redundancy as every token is encoded in place, even if the same item has already been denoted. Without any context-awareness, the allocated memory will equal to the overall counted occurrences of items corresponding to the data structures as illustrated in Figure 6.2 multiplied by their respective size, which is simply worst case. However, many rules share e.g. partially the same conditions, reference the same slots or utilize the same expressions. As long as the restrictions mentioned above are accounted for, this redundancy can be discarded during the compilation process to optimize bytecode size.

At the beginning, and as visualized in the middle part of Figure 6.3, all parts of a ruleset are parsed into a tree-based data structure with its root being the `rls_header_t` structure. From an implementation point of view, whenever two or more identical subtrees reside in the obtained data structure, the encoded information is equal, therefore redundant and subject to optimization. By merging these subtrees into one and adjusting all references accordingly, bytecode size can be significantly cut. The remaining data structure then of course loses its tree-based structure, but will still adhere to a directed acyclic graph (DAG).

Since the optimization process itself is recursive and all address adjustments have to be completed prior to image generation, the bytecode optimization is performed in two steps. Step one allocates all parsed items in their corresponding arrays as usual to gather a temporary image layout, while step two merges memory regions, calculates and re-assigns new addresses and outputs the optimized bytecode image afterwards. This simple compression scheme is not lossy, yet very effective especially for large rulesets.

Table 6.2: Bytecode sizes of selected rulesets

	Xmas	Turing	DD	GRA
unoptimized	1676 byte	2260 byte	4944 byte	6860 byte
optimized	914 byte	1110 byte	1682 byte	2004 byte
% saved	45,5%	50,8%	56,9%	70,8%

Table 6.2 gives an overview of possible savings for selected rulesets. These are freely available from the FACTS project website [136] and therefore only presented briefly here. The first, the *Xmas* ruleset implements coordinated scheduling of an LED blinking pattern on ScatterWeb sensor nodes. Therefore, a master node sends requests to light and/or turn the LED off to slave nodes, which filter these commands dependent on their position in respect to the master node, see also Appendix B.1 for the complete sourcecode. The *Turing* ruleset implements a Turing machine, developed at the very beginning of language specification to probe its capability. It comprises only four rules, one for moving along the tape, one each for handling the actions on left and right border of the tape and one for error detection.[1] With DD, an implementation of the directed diffusion routing protocol is referenced [83], which will be discussed in detail in Section 7.1. Its implementation in rules is denoted in Appendix B.2 The last ruleset presented is the *GenericRoleAssignment* ruleset (GRA), which implements the solution of a coverage problem according to the GRA approach [55]. It is noteworthy that almost all bytecode sizes can be cut in half through optimization, and, even though the size of unoptimized bytecode may be initially quite large, in comparison to those of a native implementation, see Table 7.1, these numbers account for only a fraction of the required memory in ROM to denote the same algorithm. A smaller bytecode image is benefical for reasons of more efficient OTA flashing capabilities in terms of packets sent and energy spent for retasking, as well as for mere programmability of sensor nodes in respect to encodable behavior.

Figure 6.4 clarifies how exactly the savings were achieved in relation to the data structures. Rules, facts and properties are not touched during the optimization process at all since they naturally represent distinct items, even in case e.g. two facts of the same name and properties are specified. Not surprisingly either is the low optimization capability for statements: Only in case the exact same sequence of statements is present, optimization may be applied. In both, the DD and the GRA ruleset, rules with only one statement but different conditions fulfill this criteria.

Most appealing for optimization are slots, followed by variables and expres-

[1]The bytecode sizes differ slightly from those given in [153].This can be accounted to the fact that unoptimized compilation here equals to worst case memory allocation with duplicate blocks due to copying, as well as slight increases in size due to dependency tagging.

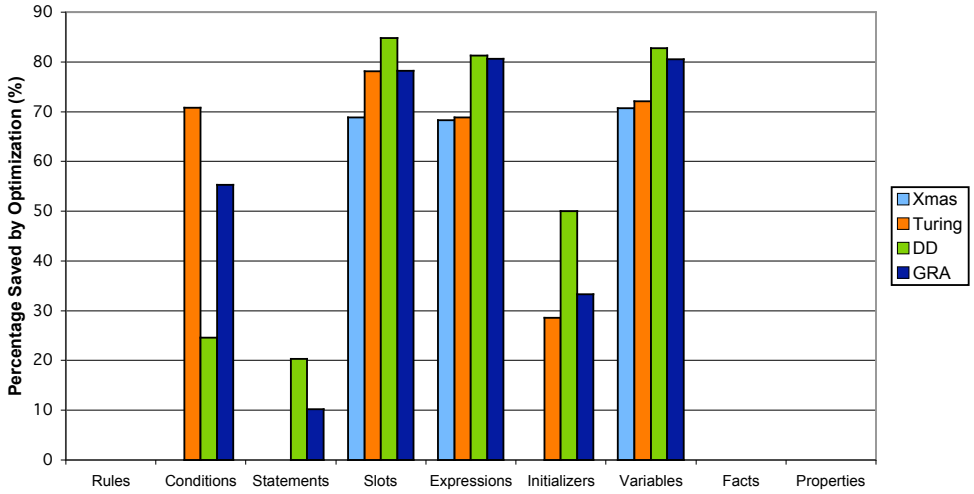


Figure 6.4: Optimization per data structure

sions, a circumstance which mirrors the expectation: Slots are filters, introduced to RDL to enable content-based addressing for facts. To facilitate reusability, slots may be named and referenced by these names in subsequent rules, adding to their repeated utilization. Likewise, variables denote the key/value tuples specified within a property, thus are very elementary parts to be accessed during programming. Expressions on the other hand are basic building blocks for conditions, statements and slots, encoding search spaces for property values and operation performed upon them. An interesting aspect is the discrepancy in values regarding optimization values achieved for conditions. While no condition has been able to be discarded in the Xmas ruleset, and roughly 25% in the directed diffusion ruleset, the Turing ruleset allows for well over 70% of its conditions to be saved. The reasons for this mixed ability for optimization become apparent when looking at the exact context where conditions are saved in the latter: These conditions are part of slot encoding, which are frequently referenced throughout the ruleset. In contrast, the Xmas ruleset offers not a single slot with a condition, and therefore no capacity for these immense savings.

6.1.3 Bytecode Optimization II: Faster Execution due to Dependency Analysis

The structure of bytecode or executables after compilation can have a tremendous influence on program execution time at runtime, a fact widely recognized within the area of compiler construction. Several techniques to speed up program execution are therefore directly applied at compile-time, such as loop unrolling to

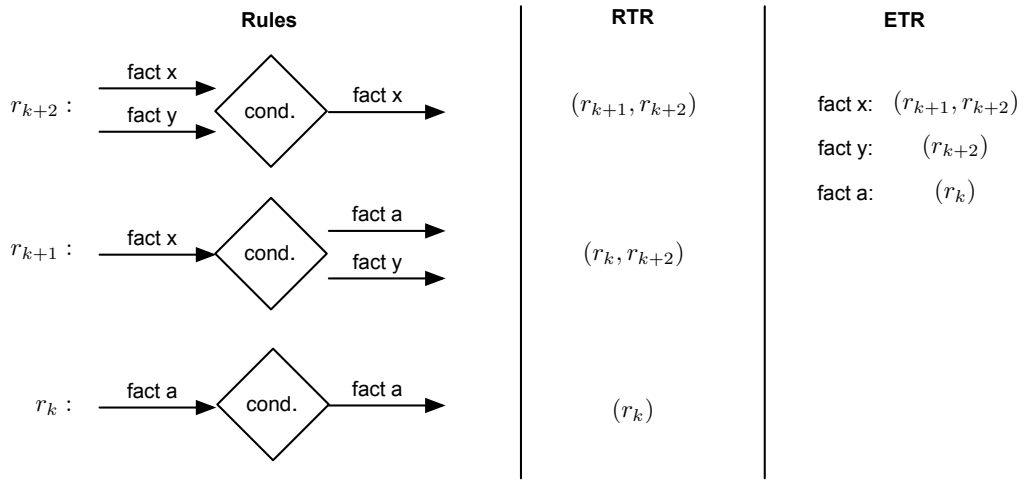


Figure 6.5: Control dependencies within rulesets

prevent speculative execution and/or branch prediction within the processor or, more particular, the VLIW instruction sets to identify instruction level parallelism (ILP).

The first step towards improved execution performance in the FACTS-rc implementation has been to sort the rules according to their priority prior to starting the compilation process. The rule engine may then pass sequentially through the bytecode for rule evaluation. In order to provide further optimization, generally two directions may be followed, runtime program analysis (such as e.g. implemented with branch prediction mechanisms for sequential ISAs) or compile-time program analysis. Due to stack memory constraints, we strongly emphasize the usage of the latter.

Despite the first impression that all rules of a ruleset seem to be relatively independent entities for data processing, a lot of information on the actual program flow is conveyed at specification time. Although the usual execution cycle for production rules is triggered by an unpredictable event, the actual production process is encoded within the control dependencies of the rules. Figure 6.5 illustrates this relationship with the help of a small example.

On the left hand side, three arbitrary rules of a ruleset, along with the facts they reference, are depicted. Note that in the following, the term *input fact* will refer to those facts that are referenced in the condition part, whereas *output facts* subsume those facts that are either defined or set in the statements of the rule as they directly influence program flow during the next production run. In case rule r_{k+2} is executed in this example, it thus alters `fact x`, rule r_{k+1} modifies the facts `fact a` and `fact y` and finally rule r_k the fact `fact a`, respectively. In the next evaluation run, the execution of rule r_{k+2} *may* thus trigger itself again

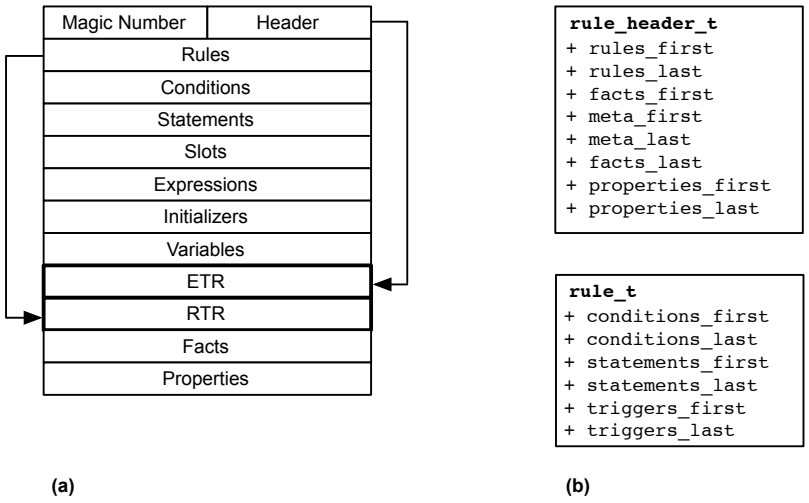


Figure 6.6: (a) FACTS runtime environment bytecode image layout after tagging compile time information (b) Revised data structures within bytecode

and/or rule r_{k+1}, rule r_{k+1} *may* trigger the execution of rules r_k and/or r_{k+2}, and rule r_k *may* trigger itself, which is displayed in the middle part of the Figure. Recall from Section 5.2 that only facts in δ_m will influence program flow in the next run. This information denoting the control dependencies between individual rules is labeled RTR (rule-triggers-rule) information. A different representation of the same relationship is depicted in the right part, subsumed under the label ETR (event-triggers-rule): Here, potential rules triggering associated with the output facts of the rules are presented.

Compile-time optimization now builds upon the observation that this knowledge about control dependencies is already available during bytecode generation. The reverse conclusion that can be drawn if this knowledge is kept, is that unless a certain rule has been executed in the previous run, another rule will not be able to trigger, thus checking its conditions is not necessary. As a consequence, the number of rules to be evaluated at runtime can be cut significantly if control dependencies are encoded within the bytecode. From a performance point of view, it is wise to encode not only the control dependencies between rules, but separately the (somewhat redundant) information on the distinct events that may trigger a rule. This ensures fast reactivity of a system to incoming event facts at the expense of slightly increased overhead in terms of bytecode size.

Implementation of the describe enhancements is straight-forward. The compiler keeps track of the input facts and the relevant output facts of a rule, and creates a data structure per rule which holds these as a list, as well as a list of rules that it is able to trigger. While passing through the source code, the list of potential rules to trigger is filled for each rule as the compiler checks for equality of

Table 6.3: Bytecode sizes of selected rulesets with compile-time enhancement

	Xmas	Turing	DD	GRA	FROMS
unoptimized	1676 byte	2260 byte	4944 byte	6860 byte	18902 byte
optimized	914 byte	1110 byte	1682 byte	2004 byte	5660 byte
ct_opt	+ 132 byte	+ 48 byte	+ 284 byte	+ 184 byte	+ 1642 byte
ct_opt in %	+ 14,4%	+ 4,3%	+ 16,9%	+ 9,2%	+ 29%

input facts of already parsed rules and output facts of the current rule in question and vice versa. In a second step, both ETR and RTR dependencies are derived and encoded as follows:

Each fact is associated with a list of rules that have to be checked upon its emergence, supplied as a start and end address of a memory region within the bytecode that contains this in-order sequence of rules. Start and end address of this ETR meta information is encoded within the header of the ruleset with four additional bytes. Likewise, the RTR relationship is denoted for every rule by supplying the start and and addresses of a bytecode section which captures references to the rules that need to be evaluated at runtime given the rule is executed. Figure 6.6 (a) shows the new bytecode layout obtained after tagging the compile-time information to the initial image, while (b) visualizes the simple addition that has to be made to the corresponding data structures. As can be seen, both data structures contain the same information, and supplying only the ETR would have been sufficient. The advantage of nevertheless encoding both data structures becomes apparent at runtime, see also Section 6.2.3. For each production run, an extra scan of the fact repository can be spared as the schedule for rule checking may be acquired on-the-fly during the previous run.

For quantitative evaluation of the effect of compile-time tagging, two issues are of interest: the incurred overhead in bytecode size as well as the performance gain at runtime. Since the second measure can only be put into context if the general rule evaluation scheme has been introduced, we will return to this in Section 6.2.3

Table 6.3 summarizes the overhead for the selected rulesets already evaluated in the last section. Relative overhead is calculated in respect to the optimized bytecode size to illustrate worst case. Besides the Turing ruleset, the relative overhead created can be roughly categorized to be around 10 - 15%. The actual size of meta information needed depends on two parameters: First of all, in case the mere number of rules increases, the probability that rules reference similar facts increases up to a certain point. It is quite frequent that independent application states deal with completely different facts, so that as a rule of thumb only a bound set of rules reference identical facts. Since the Turing ruleset has only four rules, the bytecode needed to encode the RTR section is negligible, a fact which

is nicely mirrored by the small overhead. Secondly, the number of distinct facts used to encode application behavior plays an important role and goes directly along the first influential parameter. In case a programmer uses a sole fact for state management that is referenced and altered throughout a complete ruleset, the individual rules appear extremely dependent on one another. This naturally results in a very large overhead, which will not even help to speed up execution during rule evaluation as will become evident later on.

6.2 FACTS runtime environment

The core of the FACTS runtime environment builds the rule engine with its associated fact repository. Under the supervision of the rule engine, rule bytecode supplied by the corresponding compiler backend is interpreted, incoming events are dispatched, rules are scheduled and data is acquired according to application requests. At the beginning of the chapter it has been mentioned that several target platforms for rule interpretation exist. Due to the focus of this work to provide especially practical support for wireless sensor network tasking, in the following, the emphasis will be on the rule engine implemented to run on sensor nodes at first. Since however all targets face the same language, the major difference between them is the interface design to the underlying system. Later on, simulative measures will as well be presented in Section 6.5 as they are an excellent means for pre-deployment, large scale algorithmic testing.

Literature provides a wide range of algorithms to implement rule evaluation strategies dependent on given language semantics. In case a forward-chaining approach is intended, the basic evaluation cycle can be subdivided into a matching phase of existing facts to a provided ruleset, a selection phase that determines the actual rule to be executed and finally an action phase in which the right-hand-side of the chosen rule is applied.

The most prominent approach for production rule processing, given the prerequisite of facing stateful language semantics as provided by RDL, is probably the utilization of a Rete network which supplies very efficient pattern matching capabilities [52]. Its general idea is based on the observations that within a rule base, a lot of rules share similar conditions parts, and that changes to the fact repository usually effect only a subset of all provided rules. These are then used to substantially cut the amount of rules that have to be examined in each production step, increasing reactivity of the system and execution speed. In a nutshell, rules and facts are therefore organized in a direct acyclic graph of a so called *Rete network*, where each node corresponds to a pattern in the condition part of a rule that facts have to comply to. A path from the root to a leaf represents the complete left-hand-side of a rule that has to evaluate to true before its actions may be applied. Shared conditions of rules result in shared paths within the graph. Each node simply keeps track of the facts that match its path from the root, and, in case changes are applied, moves the fact accordingly through the network. The

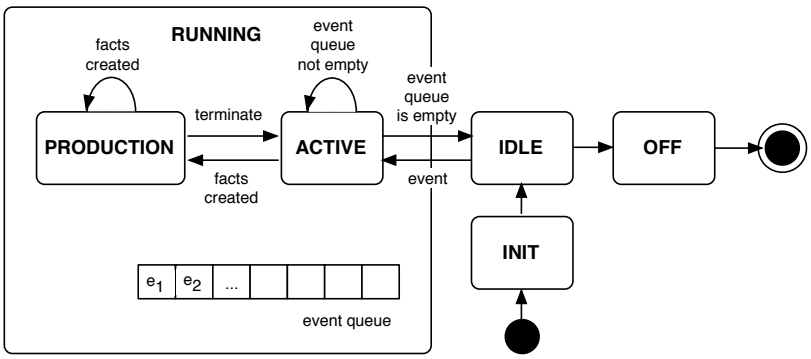


Figure 6.7: States of the FACTS rule engine

selection phase of the match-select-execute cycle is reached as soon as one or more facts arrive at the end of a path.

The major drawback of Rete, and the reason why it is not applicable on wireless sensor nodes, is that it trades memory for execution efficiency. Clearly, the complete rule base as well as at least all actively matching facts, have to be materialized for its implementation. While this is an acceptable requirement for custom-equipped computational devices, especially embedded devices suffer however from severe memory shortage. In this case, optimization goals *have* to be exactly the opposite to achieve a running system in the first place. Hence, rule engine implementation for RDL interpretation has been concerned with minimizing stack memory consumption during rule processing instead, tolerating longer execution time and sacrificing system reactivity in case memory can therefore be spared.

Mere hardware constraints however not only influence the chosen rule processing algorithm, but also have a direct impact on allocable memory for fact repository size and the size of the event buffer, and provide an upper bound on the number of supplied rules. In the following, we will therefore present the basic rule evaluation algorithm and quantify the impact of a revised algorithm utilizing the dependency information supplied with the bytecode as discussed in Section 6.1.3. A comprehensive overview of associated costs for different features implemented with the vanilla FACTS runtime environment in terms of flash and stack memory is given in Section 6.4 since this only makes sense when all features have been reviewed.

6.2.1 Basic rule processing: Strictly sequential rule analysis

Rule processing in FACTS is determined by the interplay of the rule engine, the fact repository and the rule image. Due to the sizes of the bytecode images, these have to be stored in the flash memory, while the fact repository is allocated in

Algorithm 1 Basic rule processing

```
1: for (i = header.rules_first to header.rules_last) do
2:   read rule at address i
3:   if (evaluate(rule.conditions_first, rule.conditions_last)) then
4:     applyActions(rule.statements_first, rule.statements_last)
5:   end if
6: end for
```

RAM to grant fast access at runtime. A careful utilization of stack memory by the rule engine implementation will therefore yield increased capabilities for fact storage.

From a global point of view, the FACTS-re runs as a sole user application on top of the ScatterWeb MSB430 firmware, meaning that all incoming events and callbacks are handled by FACTS. After a state of initialization, the *INIT* state, in which global variables necessary for smooth operation are read from the header of the bytecode image and facts specified to serve as constants are swapped into the fact repository, the rule engine sets itself to a state of *IDLE*. From this point on, the rule engine is operational, and will follow the interaction scheme illustrated by the state diagram in Figure 6.7

Unless an external event triggers the rule engine to wake, it will resume in *IDLE* mode, passing control to the underlying firmware. Any possibility may now be explored for energy conservation, e.g. turning the microcontroller to low-power mode or enabling wake-on-radio functionality if supplied by the transceiver, making FACTS-re also from an implementation perspective compatible to the requested event-centric processing scheme. Note that the application programmer is however nicely shielded from any of these hardware concerns by design as he is only exposed to rule specification.

Events streaming into the system are materialized as facts, put into a separate event queue for further processing and the rule engine is notified accordingly. Events comprise incoming packets, callbacks from formerly set user timers as well as sensor samples that have been supplied by returning functions. All events, or facts respectively, are tagged with a timestamp of their occurrence and sorted within the queue in an ascending manner to enforce correct temporal order of subsequent processing. As a reaction to a notification, the rule engine changes state, becomes *ACTIVE*, dequeues the first event from the event queue to store it within its fact repository and begins rule evaluation. The basic processing scheme is denoted in Algorithm 6.1, and follows a very simple *fetch-resolve-execute* cycle.

In this evaluation variant, *all* rules are checked sequentially. Since on the one hand, stack memory has to be used carefully, and to ensure limiting read access to flash to only relevant elements on the other, rule parts are fetched solely on demand. Scanning all rules (lines 1 and 2), their conditions are read one by one and checked whether a fact matching the supplied pattern resides in the fact repository. As soon as a condition fails to evaluate to true, lazy evaluation is

applied and the rule engine jumps directly to the next rule encoded within the image. In case all conditions of the current rule evaluate however to true, and given the circumstance that one of the involved matches is a new event, all actions of this rule are scheduled for execution (line 4). This loop terminates when all rules have been successfully checked and statements applied if required.

Three options are then available for continuation: First of all, the run of the rule engine may have spawned new facts. Then, the rule engine will turn from the *ACTIVE* state to the *PRODUCTION* state, setting all previous event facts to regular facts, and all newly added facts to serve as events within the fact repository. This does *not* effect the event queue at all, as this is a separate data structure shielded from the internal workings of the repository. Indeed, all events recognized concurrent to processing are stored in this event queue, which is however completely independent of the current execution. Afterwards, rule evaluation is triggered from the beginning until the production run cycle terminates and the envisioned reaction to the triggering event is accomplished. The rule engine will return back to *ACTIVE* mode.

Secondly, no new facts may have been added to the fact repository, but during ruleset evaluation new events may have occurred. In this case, the next event, which corresponds to also the next event from a temporal perspective, is dequeued for adequate processing. A third possibility is that neither a production run is necessary, nor are events available that request reactions. The rule engine may then switch to *IDLE* mode. It has proven useful to have a mechanism for manual termination of a rule engine at runtime, e.g. if during a deployment a node exhibits faulty behavior for unknown reasons. A command, which is a means to invoke functions via the serial interface provided by the ScatterWeb firmware, is therefore implemented for FACTS to stop the engine. In state *OFF*, all registered timers are removed and no events forwarded to the rule engine, prohibiting any processing.

The distinction between production and reaction is more than an implementation detail, making it necessary to point out their semantical difference: For occurring events, it is mandatory to ensure strict temporal order in order to support complex event processing schemes and their correct evaluation as specified by applications. However, facts produced as a matter of rule execution do not have such a temporal aspect to them. Therefore, materializing these facts within the fact repository at creation time is perfectly feasible. Or, to put it in other words, the fusion of the ECA world with the production rule aspects within FACTS becomes apparent right at this point: Event-processing is combined with production rule semantics in one, comprehensive model.

6.2.2 Pattern matching in detail

So far, the above sketched rule engine states reveal no details on how the actual processing is implemented. In order to enable fast understanding, this is best illustrated with the help of an example, an excerpt from the `Xmas` ruleset, see Listing 6.1.

Listing 6.1: Excerpt from the `Xmas` ruleset.

```
1  ruleset Xmas
2  [...]
3  fact system [broadcast = 0, tx-range = 10]
4
5  slot systemBroadcast = {system broadcast}
6  slot systemTxRange = {system tx-range}
7  slot systemID = {system owner}
8
9  rule getNumRows 100
10 <- exists {bootstrap}
11 -> define row [depth = 0]
12 -> define light [ON = true, node = 0]
13 -> send systemBroadcast systemTxRange {row}
14
15 rule getMaxNum 99
16 <- exists {row_reply
17     <- eval ({this depth} > {row depth})}
18 -> set {row depth} = {row_reply depth}
19 -> retract {row_reply}
20 -> call removeTimer ({delay})
21 -> call setTimer ({delay}, 1)
22
23 [...]
24
25 rule replyDepth 80
26 <- exists {row}
27 <- eval (systemID != 1)
28 -> define row_reply [depth = systemID]
29 -> send 1 systemTxRange {row_reply}
30 -> retract {row_reply}
31 -> retract {row}
```

Generally, these three rules implement a simple interaction scheme between nodes within a network. A mastering node, marked with id 1, requests to find out the maximum depth of the network by sending out a `row` fact (line 13). In response, a set of slave nodes react by replying with their own id (line 29), copied to a `row_relply` fact (line 28). Whenever a `row_relply` fact is recognized at the master node, note that `row_relply` facts are sent in a unicast manner, it will evaluate whether this new fact‘s annotated depth is larger than the depth currently cached in a `row` fact (lines 16 and 17) and eventually update the fact property.

The interesting part is the resolution of slots as pattern matching can become arbitrarily complex. Recall that syntactically, slots may either be declared en block for subsequent usage, e.g. the slot `systemBroadcast`, or denoted within the rule itself, which however does not influence the evaluation process.

The simplest case of matching is when only a name of a slot is provided which is

the case for the condition specified in line 10. The only requirement for triggering rule `getNumRows` is the availability of an event fact named `bootstrap` in the fact repository. Worst case, the rule engine will have to run through the fact repository once for slot resolution. A similarly easy operation is the evaluation of the second condition of rule `replyDepth` (line 27). Here, the property `owner` of a `system` fact has to be checked and differ from one, an operation which involves additional fetching of the fact's properties for comparison.

As soon as slots exhibit patterns involving additional conditions, the reference fact for condition evaluation has to be marked, which is done by a programmer with the keyword `this` (line 17). All subsequent comparisons will therefore refer to the current fact. From an implementation point of view, the context of the current slot resolution has to be stored, and condition evaluation started by retrieving the right-hand-side of the condition, strictly avoiding self-reference. All in all, the process of evaluating the different language elements, relating them to each other and keeping track of the context they reside in, is recursive, comprising quite some effort to keep correct rule engine state.

Regarding stack memory, sooner or later the rule engine may nevertheless run into a problem. This circumstance is owed to the fact that the RDL grammar allows for infinite nesting of slots, conditions and expressions. Below, this problem is illustrated with the help of a possible production derived from the RDL grammar, which is for reasons of brevity displayed in a simplified manner, see also Section 5.1.

slot	→	identifier | name [key] [condition]
condition	→	‘←‘ ‘exists‘ slot | ‘←‘ ‘eval‘ (expression comp expression)
expression	→	variable | (unary_op expression) | (expression comp expression)
variable	→	bool | integer | string | slot
slot	⇒	name [key] [condition]
	⇒	name [key] ‘←‘ ‘eval‘ (expression comp expression)
	⇒	name [key] ‘←‘ ‘eval‘ ((unary_op expression) comp variable)
	⇒	name [key] ‘←‘ ‘eval‘ ((unary_op (expression comp expression)) comp slot)

Practically, this infinite nesting cannot be supported since stack overflow will then crash a node at runtime. The precautions taken to prevent this scenario is to cut recursion at a reasonable depth: Binary expressions are not eligible to be nested within other binary expressions, while slots referred to within expressions may not exhibit conditions. At first sight, this may seem very restrictive, especially in the

Algorithm 2 Rule processing with available dependencies: Reaction to event

```
1: given the event that triggers the reaction
2: for (i = header.meta_first to header.meta_last) do
3:   read ETR information at address i
4:   if (ETR fact name matches event) then
5:     for (i = etr.rules_first to etr.rules_last) do
6:       read rule at address i
7:       if (evaluate(rule.conditions_first, rule.conditions_last))
         then
8:         applyActions(rule.statements_first, rule.statements_last)
9:       end if
10:    end for
11:  end if
12:  break for loop
13: end for
```

latter case, since e.g. a `set` statement can then not filter conditional patterns for their right-hand-side of the assignment. It does however not effect the general expressivity of the language. The problem can be simply bypassed by specifying a temporary fact for binding an arbitrarily complex slot in question to a property, which may then be easily referenced within the expression.

6.2.3 Rule processing utilizing compile-time dependency analysis

The rule evaluation and pattern matching scheme presented above requires recurrent resolution of patterns provided within the bytecode image against the current state of the fact respository, which is a costly procedure in terms of evaluation time. Optimization strategies can either tackle this problem at runtime by means of caching valuable information such as resolved slots, indexing the fact repository for better access or structuring the evaluation path as e.g. done within a Rete network, or at compile-time preventing unnecessary rule evaluation cycles in the first place. Since available RAM prohibits extensive exploration of the first approach, we rely on the second option.

In Section 6.1.3 the motivation of supplying additional information on inter-rule dependencies has already been presented. Hence this section is dedicated to the actual utilization of the available dependency information at runtime, and the effect this enhancement has on the overall performance of the rule engine.

One important difference to the basic rule processing scheme is that dependent on the current state the rule engine resides in, separate regions within the bytecode image are consulted for building the actual evaluation schedule. In the basic scheme, this schedule subsumes all rules. In case compile-time information is available, a fact that is denoted within the header of the image and thus recognized by the rule engine at initialization time, an incoming event will activate

Algorithm 3 Rule processing with available dependencies: Concurrent scheduling of productions

```
1: given a queue q of addresses of rules to check
2: j = sizeof(q)
3: for (i = 0 to j) do
4:    a = dequeue_first(q)
5:    read rule at address a
6:    if (evaluate(rule.conditions_first, rule.conditions_last)) then
7:       applyActions(rule.statements_first, rule.statements_last)
8:       enqueue(rtr.rules_first, rtr.rules_last)
9:    end if
10: end for
11: remove duplicate rule addresses from q
12: sort q
```

scanning the ETR region of the bytecode. The corresponding algorithm is denoted in Algorithm 2 of this section.

This region contains the addresses of all rules that may possibly be triggered by the dequeued event in the correct order for direct evaluation. Since the rules referenced may only be a small portion of the ruleset, the associated `for` loop (lines 5 - 10), may as well iterate over considerable less items for evaluation. Note that with cutting the number of rules to be evaluated, the resolution of all corresponding rule elements within the condition part can also be spared.

If the rule engine is executing productions, thus resides in *PRODUCTION* state but also in the first run following an event trigger, the successive schedule is acquired during the current execution, presented in detail in Algorithm 3.

This strategy exploits the fact that each rule is automatically tagged with references to a bytecode region which encodes all rules its execution can invoke in the next production run. Therefore, a queue that will later on comprise the next schedule, is filled with the corresponding rule addresses after all statements of the triggered rule are executed. Within one run through the rule base, several rules may fire, thus enqueue their respective follow-up rules. Duplicates have to be removed and the rules sorted according to their priority before restarting the evaluation run. The vigilant reader may recall that rule priorities are not explicitly denoted within the encoding scheme, and thus wonder how this sorting can take place. However, it has to be pointed out, that the same information is conveyed by the ordering of the rules within the bytecode image, and thus sorting can be performed upon the cached addresses.

6.2.4 Impact of utilizing compile-time dependency analysis on engine performance

In order to evaluate the impact of compile-time analysis on system performance, two representative rulesets have been chosen, namely the afore mentioned `Xmas` ruleset and the `DD` ruleset implementing a directed diffusion variant. The reasons for taking these ones is that their results nicely mirror the spectrum of achievable performance gain, while at the same time the incurred overhead in terms of additional image size is roughly the same (14.4% vs. 16.9%). In the following, we will first discuss the results of the `Xmas` ruleset execution, then turn to the `DD` ruleset and provide an explanation for the observable differences afterwards.

For running the `Xmas` ruleset, a network of ten nodes has been utilized. The application itself executes in rounds, each of which subsumes a master node requesting all slave nodes to enlight their LED following a predefined, temporal pattern it specifies, and then turn their lights off upon demand. Overall, the ruleset comprises 9 rules, 22 conditions, 20 slots and 20 expressions. The diagram depicted in Figure 6.8 plots the average number of language constructs such as e.g. rules, conditions and slots evaluated for both master and slave nodes separately after five rounds from three runs of the experiment. Naturally the highest burden is put on the master as the central coordinator, thus the numbers of evaluated elements differ substantially.

Without available compile-time enhancements, all slave nodes run 20 times through their ruleset, thus evaluate exactly 180 rules, about the same number of conditions and resolve 273 slots. These numbers can be significantly cut to roughly 40 rules, 50 conditions and about half the number of slots with the availability of compile-time information (slaves + ct). Regarding expressions, no improvement can be obtained. All expressions that affect slave nodes are part of the statements of rules, thus are equally resolved in both cases.

An even better result can be achieved for the master node: about 150 rule evaluations compared to initially 500, a little less improvement regarding conditions (363 against 221), almost a third of the number of slot, and more than half the number of expressions can be spared with the help of dependency information.

Although these values provide a first impression on achievable performance gain, they nevertheless hide the overhead of having to fetch and decode the dependency information from the image. Since processing timespans are so short that their exact measurement turned out to be difficult, we measure performance gain in total number of bytes read from flash to RAM for evaluation purposes. This is a reasonable metric as flash access time predominantly contributes to overall execution time. The corresponding results, average number of bytes read shown on the y-axis and broken down to individual rounds on the x scale, are presented in Figure 6.9.

For comparison purposes, round five is especially interesting: For the slaves, that in regard to the master face only a fraction of the processing load, the actual savings in terms of bytes read range around 30%. The overhead for reading the

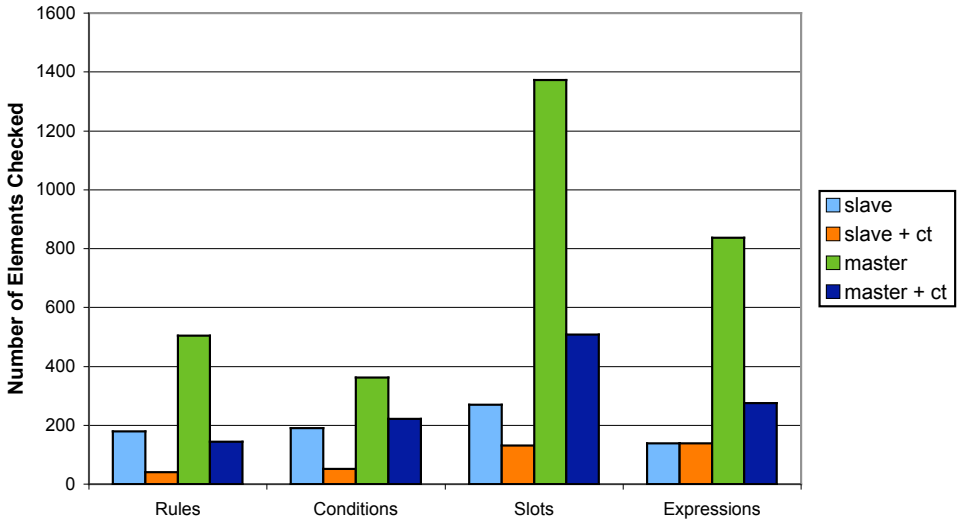


Figure 6.8: Number of elements checked at runtime dependent on role and compile-time dependency analysis (+ ct) (Xmas ruleset/5 rounds)

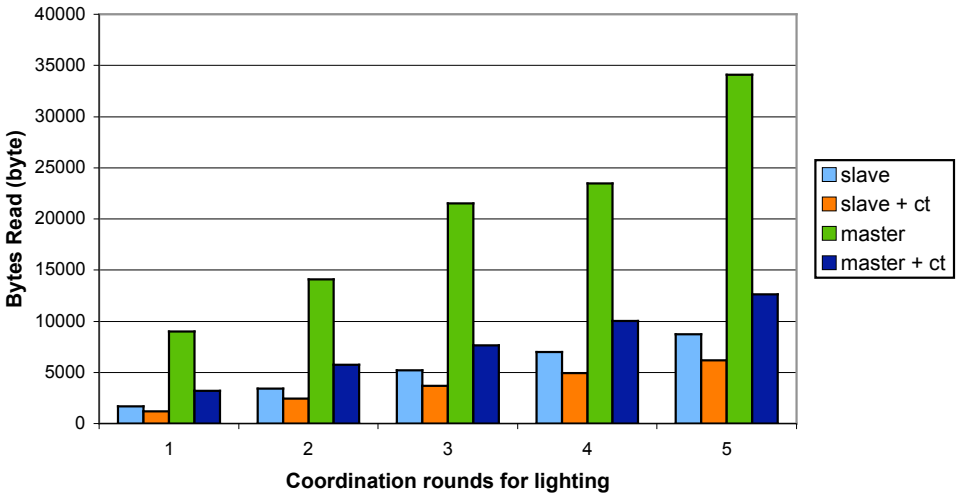


Figure 6.9: Total number of bytes read at runtime after each round for different role and compile-time enhancement (+ ct) (Xmas ruleset)

Table 6.5: Percentage saved (Xmas ruleset/5 rounds)

	Bytes read	Rules	Conditions	Slots	Expressions
regular execution	11254 byte	212.4	207.3	381.4	208
+ ct analysis	6813 byte	51.4	68	170	151.8
% saved	39.5%	75.8%	67.2%	55.5%	27%

additional compile-time information has indeed a significant impact in this case, with performance gain stabilizing after a few rounds to a constant factor since processing during network initialization is also part of the displayed numbers. The master node can benefit from the compile-time information even more: Less than half the number of bytes (34kbyte vs. 12.6kbyte) have been swapped after the execution of five rounds.

To sum up the results, Table 6.5 once again highlights the average benefit of applying and utilizing compile-time dependency analysis with respect to the entire network. All values are calculated by averaging obtained results after five rounds, independent of the role nodes take within the network. With an overall number of almost 40% less bytes read from flash, the effectiveness of the compile-time analysis is not questionable. Conforming to the expectations, an impressive number of rule evaluation and condition resolution can be cut and thus contribute significantly to the achievable performance gain.

Results for the `DD` ruleset shed a slightly different light on the impact of compile-time analysis. Once again, a network of ten nodes utilizing the topology depicted in Figure 7.3 has been flashed with the appropriate ruleset, featuring about twice the elements as have been available in the `Xmas` ruleset - 18 rules, 46 conditions, 34 slots and 35 expressions. Within the network, nodes can either have the role of being a sink, thus issue requests for data, act as routers for both requests and reply facts, or function as a data sink. The exact network layout and executed algorithm will be discussed in Section 7.1. Similar to the above, results are displayed plotting number of elements checked per type and role of the node in Figure 6.10 and number of actually read bytes (this time in kbytes) against role in Figure 6.11.

The values have been obtained running the algorithm for approximately 100 seconds. In this time, the source node sent out 20 packets towards the sinks, while these distributed between five and six route maintenance packets. Numbers have been acquired in three distinct experiments and averaged afterwards.

Savings for rules and conditions range for sinks and sources well above 50% and also slots and expressions, although not as overwhelming as in the `Xmas` ruleset, confirm the general observation of high saving potential as in the previous data set. Processing load put on the nodes that conform to the role of routers is with respect to the other nodes higher, but unfortunately the impact of compile-time information not as prominent as in the other cases. Note that the increase in

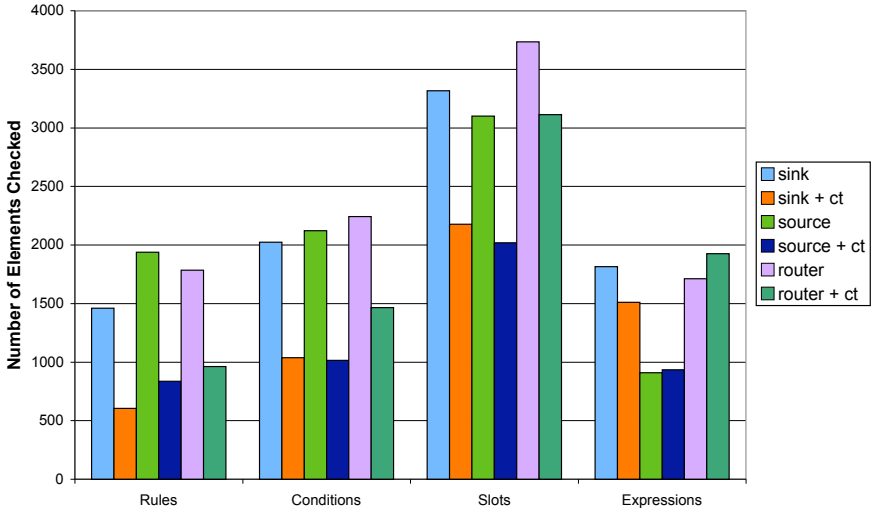


Figure 6.10: Number of elements checked at runtime dependent on role and compile-time dependency analysis (DD ruleset/100 seconds)

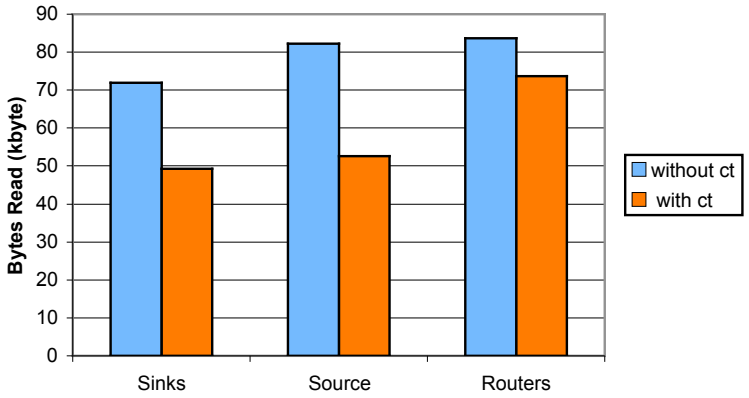


Figure 6.11: Total number of kbyte read at runtime for different role and compile-time enhancement (DD ruleset/100 seconds)

Table 6.6: Percentage saved (DD ruleset / 100 seconds)

	Bytes read	Rules	Conditions	Slots	Expressions
regular execution	79.3 kbyte	1726.5	2128.1	3382.9	1477.8
+ ct analysis	58.5 kbyte	800.1	1171.2	2435.8	1455.4
% saved	26.2%	53.7%	44.9%	28%	1.5%

expression evaluation utilizing compile-time dependencies can however not be attributed to the optimization scheme itself, since by no means additional language elements can appear for resolution. Rather, this mirrors the fluctuation within the routing scheme, and probably a tendency to slightly longer routes in the corresponding experiments. As a result of an increased number of hops between source and sinks, the overall number of statements, and thus expressions, executed for routing decisions will increase equally.

Turning to the chart which displays the average, total number of bytes read from flash, the above presented differences become even more apparent. Although for each type of network node, a utilization of compile-time dependency information is able to lower the cost for read access to flash, the actual gain for routers is with approximately 12% rather low, while the benefit for sinks and sources can be identified to range between 30% to 40%. From a network perspective, Table 6.6 once again summarizes average savings in percent.

Despite the fact that the attainable impact of compile-time enhancements with dependency information is much higher for the `Xmas` ruleset, the measurements obtained for the `DD` routing ruleset still attest this strategy an impact that without doubt legitimates its utilization.

A look at the actual rules reveals the reason for the different optimization capabilities of the given rulesets. Only a third of the rules that are denoted in the `Xmas` ruleset target slave node behavior, and associated facts solely affect these rules upon rule execution. As a consequence, the compile-time information is able to restrict the evaluation space for reactions and productions to a very small subset. At the other extreme, facts that are involved in storing and updating routing information are referenced throughout the `DD` ruleset, thus have a very high visibility. Any change to this routing information will therefore yield scheduling of almost all rules, a circumstance that in response requires swapping a lot of compile-time information from flash as well as subsequent evaluation. The price paid in additional flash access to read the optimization data does in this case not pay off by cutting the number of rules to check. Recall that this is *not* the usual case, since then no optimization could be measured.

6.3 Interfacing the Sensor Node Hardware

So far, the focus of this chapter has been to illustrate how rules are compiled to bytecode and then interpreted on a wireless sensor node. The essential question how data is obtained, thus how exactly mandatory access to a sensor node's hardware capabilities is integrated in the process of rule evaluation has not been touched yet and is subject to this section. Sensors, possibly actors or other devices of the user interface (UI), peripheral hardware such as secondary storage or a serial interface and, as has become apparent in Section 5.3, the possibility to configure timers have to be made available to a programmer.

Exporting a reasonable interface to system-related settings to applications is however a non-trivial task in middleware design: Clearly, many applications developed for wireless sensor networks demand for flexible adaptations to control the available hardware at runtime which prohibits a static compile-time configuration. Parameters such as sampling frequencies, actuation timing or thresholds for data filtering have to be able to be individually altered based on the state of a running application. Furthermore, as soon as the application development process goes from a state of prototyping to application fine-tuning, a more elaborate way of hardware configuration may be needed than at earlier stages. While during prototyping e.g. energy-efficiency is negligible it will well be of interest in a final deployment. Thus, the granularity of system access may vary over the development time of an application. This trade-off of supporting a high level of abstraction from system-related parameters to enable fast prototyping while at the same time allowing access to low-level hardware settings for application fine-tuning is addressed in this section.

Dependent on the usage pattern for system access, several options at different layers of the system architecture are available to schedule hardware- or system-related demands appropriately are available: Recall from the preceding chapter that the lowest level of abstraction is inherent to the RDL language, which offers an interface to arbitrary system functions via `Call` statements to export part of the native API where required. This way, full control of hardware settings is guaranteed whenever necessary. Figure 6.12 depicts the relationship between the FACTS middleware framework, firmware and sensor node hardware. In terms of hardware being subject to integration into RDL, the ScatterWeb MSB430 platform comprises a set of sensors, some mounted on the core module while others reside on an optional sensor board, a set of UI devices such as the LEDs, the serial interface, the hardware timer and a slot for SD card integration.

To provide a better handle for hardware configuration and control at a higher level of abstraction, so called `context_facts` have been added to the API of FACTS. Dependent on the hardware resource, a corresponding `context_fact` aggregates configuration parameters and/or encapsulates its operational mode. As a consequence, a modification to a `context_fact` will result in the adaptation of its represented resource, giving an application programmer an easy-to-use abstraction to underlying hardware.

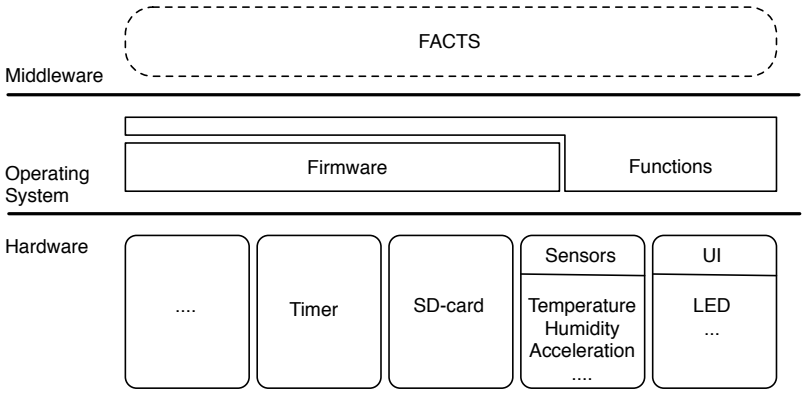


Figure 6.12: System properties to be integrated into FACTS

6.3.1 Language-inherent System Access

As mentioned above, `Call` statements are the basic access mechanism to hardware and system-related parameters that FACTS provides. A thin layer of software in between firmware and FACTS is necessary to tunnel firmware functions and make them callable from RDL rules. Besides, a set of utility functions has also been added to the `Call` API to grant support for additional low-level concerns.

Whenever such a `Call` statement is invoked, the corresponding action will be executed once. The signature of a `Call` statement is not fixed, but depends upon the called function to allow for adequate parameter passing. Return values will be supplied as facts and pushed into the fact repository. Listing 6.3 provides an overview of fact names and property keys for these system facts, whereas the interfaces for calling ScatterWeb functions are denoted in Listing 6.2.

The perception application programmers have towards hardware resources varies w.r.t. the different devices: Some, namely sensors and actors, need to be able to be configured in full extent by the application itself and settings may even vary over execution time. Hiding parameter adaption within the middleware for these resources is hence counterproductive, if not even harmful. Then again, those devices that we subsume under the label *peripherials* in the following are not subject to application-level tuning. Rather than being interested in their individual configuration, developers request these devices to simply work. As a consequence, application concerns can well be shielded from annoying configuration issues, and interface design thus resort to the easiest utilization scheme possible.

Access to Sensors and Actors

Upon issuing a sampling request to the sensor node, a fresh reading is taken. The return value will be asynchronously pushed into the fact repository as soon as the called firmware function returns. All functions representing sensor hardware

Listing 6.2: FACTS system call API.

```
1  //access to sensors
2  Call getTemperature([int res], [int mode])
3  Call getHumidity([int res], [int mode])
4  Call get3DSample([int sensitivity])
5
6  //access to UI components
7  Call setLED([int mode], [int blink_times])
8
9  //access to peripheral hardware
10 Call printFact(name fact_name)
11 Call logFact(name fact_name)
12
13 //access to timer
14 Call setTimer(name t_name, int interval)
15 Call removeTimer(name t_name)
16
17 //access to arbitrary functions
18 Call getRandom()
19 Call getPosition()
20 Call getSysInfo()
```

can be called without arguments. In case no explicit information is supplied else wise via `context_facts`, see Section 6.3.2, FACTS will simply default to lowest resolution and accuracy.

The ScatterWeb MSB430 platform, used for a prototypical implementation of FACTS, currently offers three different sensors, a temperature, humidity and an acceleration sensor, and one LED with additional sensors and actuators attachable on demand. Temperature and humidity readings can be sampled at a high or a low resolution, with a high resolution taking about four times as long. Furthermore, the humidity reading may be corrected by a temperature coefficient to gain better quality, whereas the temperature reading may be represented in degree Celsius or Fahrenheit which can be both controlled via the second property passed as a parameter. The accelerometer allows for setting its sensitivity according to its sampling domain, thus dependent on the application specific range of values that are expected to be measured. The more sophisticated an application has to be about its input values to enable decisions on application-relevant states and its output values to precisely control the sensor node behavior, the more important low-level configuration of the corresponding measurement hardware becomes - a circumstance which is nicely reflected by the chosen access scheme.

Access to Peripheral Hardware

Peripheral devices, i.e. devices which offer a kind of background service such as logging or printing debug information, should be accessible in the easiest manner possible. In contrast to sensors and actors, explicit configuration is neither needed

Listing 6.3: System facts that are returned when invoking system calls.

```
1  //sensor values
2  fact temperature [int value]
3  fact humidity [int value]
4  fact acceleration [int x, int y, int z]
5
6  //timer output
7  fact t_name
8
9  //utility function output
10 fact random [int value]
11 fact position [int x, int y]
12 fact init [bool acceleration, bool temperature, bool logFact]
```

nor desired in their case. There is for instance not only no need to let applications decide on the size of swap memory allocated to enable data logging on flash. Potential memory shortage at runtime in case overly optimistic setting have been chosen is instead a good reason to prohibit such low-level intervention.

FACTS implements mechanisms to log facts to an SD-card and to print facts e.g. for debugging purposes using the serial interface. The interface to these peripheral devices reflects the above mentioned design rationale: logging or printing using the `Call` statement is possible by specifying a name of a fact or a slot as an argument without any additional configuration options. This content-based utilization scheme mirrors the overall system design of the FACTS platform, exporting straight-forward usage semantics to the application developer.

Access to System Functions

On the software side, `Call` statements can be used to access a subset of the interfaces supplied by the ScatterWeb API as well as additional utility functions. To keep the core framework as lean as possible, the number of implemented functions is reduced to a very small set and comprises mechanisms to handle software timers, for creation of random values and to support self reflection on a sensor node. However, in case specific applications require additional, native functionality, this interface can be easily adapted and further, sophisticated processing schemes such as e.g. complex, mathematical functions to aid in localization may be added on demand.

As has been discussed extensively in Chapter 5.3.1, timers take a special role during application development as they enable manual flow control, even within the event-centric world of FACTS. Since any action has to be triggered by an event, or more precisely by the occurrence of a new or altered fact, the timer interface is the only possible source for explicit, delayed event generation. Consequently, the expiration of a timer will generate a new event fact with its name being passed as an argument to the corresponding function.

Listing 6.4: Periodic sampling of the temperature sensor.

```
1  name trigSampling = "trigSampling"
2
3  rule defineTrigger 100
4  <- exists {"start"}
5  -> call setTimer(trigSampling, 30)
6
7  rule sampleTemp 99
8  <- exists {trigSampling}
9  -> retract {trigSampling}
10 -> call getTemperature(1, 0)    //high resolution, degree Celsius
11 -> call setTimer(trigSampling, 30)
```

Sensor nodes do not necessarily have to be equipped in a homogenous manner. Within a deployment, secondary storage or the additional sensor board are often mounted to only a subset of nodes to cut overall setup costs. To circumvent the case that an application developer is forced to write different programs dependent on the hardware setting, each node is enabled to dynamically query its hardware capabilities at runtime. Upon the invocation of the `getSysInfo()` or `getPosition()` function, either a fact reflecting a node's available resources or a fact that outputs position information if available is created and may then be used for subsequent matching and conditional execution of software parts.

6.3.2 Context-aware Configuration and Control

Tunneling native functions and making them accessible in RDL is perfectly enough to obtain operational rule programs. However, especially applications that make excessive use of their hardware capabilities suffer from bloated code due to the separation of configuration and control concerns that the event-driven model exhibits. A simple, periodic sampling request as denoted in Listing 6.4 illustrates this concern:

Given an arbitrary, triggering event, represented by a fact named *start* in this example, periodic sampling of the temperature sensor at a high resolution is requested. To initialize a control loop that captures this periodic behavior, a timer needs to be defined to fire an adequate event, a fact called *trigSampling* in this case (line 5), which is created upon timer expiration after 30 seconds. The rule *sampleTemp* is the actual core to continuous data acquisition as it reacts to the occurrence of a *trigSampling* fact by issuing a call to sample the temperature sensor (line 10) and setting a new timer to produce a *trigSampling* fact after 30 seconds. Setting the operational mode, thus configuration of the sensor (high quality sampling) and the sampling interval, thus its control loop, is split into individual steps and have to be individually addressed for every sensor, and possibly every actuator that is operated in a cyclic manner. As these settings are usually subject to change dependent on the application state, special attention has to be paid to adjust *all*

Listing 6.5: `Context_facts` for configuration.

```
1  //context_facts to configure the sensors
2  fact temperature_context [int samplefreq, int res, int mode]
3  fact humidity_context [int samplefreq, int res, int mode]
4  fact acceleration_context [int samplefreq, int sensitivity]
5
6  //context_fact to specify timers
7  fact timer_context [int interval, name fact_name]
8
9  //context_facts for peripherals
10 fact log_context [name fact_name]
11 fact print_context [name fact_name]
```

parameters according to the current situation. One can easily deduct that the expression of periodical tasks is not optimal when relying on language-inherent system access.

In contrast to sensors and actors, peripheral components exhibit an orthogonal utilization scheme as they are predominantly operated in direct response to application semantics. For instance, sampled data may only be relevant for logging to flash in a specific application state inherent to the application. In this case, not the control of application flow, but the filtering mechanisms lead to bloated source code: RDL offers by design no straight-forward approach to address a single fact, but filters facts via matching. As a consequence, all facts that currently reside in the fact repository and match the requested slot for logging will be written to flash when invoking a `Call` statement, even if they have been logged beforehand, but need to reside within the fact repository e.g. to allow for later on aggregation.

To overcome all these difficulties, provide an easy-to-use abstraction and enable quick, context-aware hardware reconfiguration, we added the concept of `context_facts` to the FACTS API. All configuration parameters relevant to operate sensors and actors, to define user timers or to use peripherals are assembled in particular `context_facts` for the specified component. Added to the fact repository just as a regular fact, the rule engine will configure the requested resource transparent to a programmer and according to its operational mode. The advantage of using specialized facts to integrate configuration and control is that it is a simple augmentation of the framework while at the same time preserving the overall unified data abstraction. A nice side-effect is that the transmission of `context_facts` opens up a new means for local interaction between nodes, as dynamic remote tasking can be easily implemented.

Usage of `context_facts` to encapsulate control loops

When utilizing a rule-based programming language such as RDL, the execution of a rule demands for a prior event to trigger it. As has been extensively discussed beforehand, implementing a scheduled task therefore requests a programmer to

Listing 6.6: Periodic sampling of the temperature sensor with a `context_fact`.

```
1 rule defineTrigger 100
2 <- exists {"start"}
3 -> define temperature_context [30, 1, 0]
```

manually craft the triggering event with the help of a user timer to gain control of program flow. Access to resources that are subject to periodic scheduling, see Listing 6.4, has to be enclosed within a dedicated control loop.

A generic abstraction to hide this control loop and allow for cyclic fact generation is the `timer_context` fact, see Listing 6.5. Whenever a `timer_context` fact is added to the fact repository, the rule engine will schedule a new, supervised timer. As soon as the amount of time denoted in the interval property of the `timer_context` fact has passed, a fact with the specified name in the *fact_name* property is created and added to the fact repository. In case the matching `timer_context` fact still resides in the fact repository, the timer is again set and the operation repeated until the `context_fact` is retracted. Modifications to the interval of a `timer_context` will automatically lead to timer adjustments right at alteration time, whereas a changed name of the expiration fact equals to retracting the old `timer_context` and adding a new.

Usage of `context_facts` to aggregate configuration parameters

Convenient configuration *and* control can also be achieved for sensors and actors with the help of `context_facts`. Facts for this category of devices combine the ability to denote control information with the possibility to specify configuration parameters, and aggregate these into one fact per component. Similar to the interval property for `timer_context` facts, the samplefreq property displays inquiry intervals, whilst configuration parameters equal those available in the basic `Call` statements. Since the names of the facts that contain sensor measurements are well-known within the language, explicit declaration of return values is not needed. Both, configuration or control properties can be left unspecified so that either function can be used by itself: The omission of the sampling frequency renders the `context_fact` to be a source of configuration parameters which will be applied whenever a `Call` statement without parameters is invoked. Missing configuration parameters in the contrary will allow for periodic utilization at default settings.

Unlike `timer_context` fact specification, only one `context_fact` per resource is allowed. The notion here is that the `context_fact` represents its resource as configuration parameters are explicitly provided. However, this does not prohibit resource multiplexing: An issued `Call` statement will always be executed with the settings that it promotes, yet the declaration of an additional `context_fact`, e.g. by a different application running on a node, will overwrite previous settings due to the global scope of the fact repository and global visibility of `context_facts`.

Listing 6.6, a revised implementation of periodic sampling presented in Listing 6.4, makes use of the `context_fact` for temperature sensor configuration and control to illustrates the gained features. In this particular example, lines of code can be cut to almost a quarter in regard to the previously needed.

Usage of `context_facts` for semantic timing

Logging facts to flash or printing them for debugging purposes with the `Call` statement turned out to be of a peculiar semantic during application development. In order to prevent matching a multitude of facts that bear the same name, the filtering has to be very precise, an effort that feels orthogonal to its purpose. The usage of these peripherals has more a flavor of a background functionality concurrent to the application than being an active part of it. Therefore, the introduced `context_facts` for these resources overcome the challenge of what we refer to as semantic timing: They basically realize a publish and subscribe mechanism for specified facts to aid in the application development process rather than to hinder fast prototyping.

Key to understanding their operation is that, in accordance to the event-driven model, any fact of interest for *constant* logging or printing will be a fact that has faced change during the last rule evaluation run. For instance, given the availability of secondary storage as log space, the declaration of a `log_context` fact will instruct the rule engine to log all facts that match the denoted name (which may of course represent a slot, thus encapsulate arbitrary filtering mechanisms) *and* have been added or altered. A copy of these facts is stored to a buffer page for later eviction to flash. This way, a time series of modifications to a fact will also result in a time series of logged facts on the SD-card, a circumstance that is valuable to interpret system behavior during debugging or to follow data evolution in offline analysis when recording values. Once again, `log_context` and `print_context` facts are evaluated individually at each rule iteration so that modifications e.g. to filtering parameters will directly have the desired effect. Naturally, the programmer is not restricted to the specification of a single `context_fact` as has been the case when configuration settings are inherently provided, but may consult application needs. `Call` statements are however completely oblivious to the existence of `context_facts`; duplicate logging or printing due to multiple requests can therefore not be prevented.

It is noteworthy to point out why this way of subscribing for facts can be interpreted as an action concurrent to application execution. Imagine a `print_context` fact expressing its interest for acceleration facts without any filtering. Although a rule that issues the `print_context` fact may e.g. only trigger on acceleration facts of a high intensity, subsequent printing will not.

Usage of `context_facts` for remote tasking

Syntactically, a `context_fact` is a regular fact, tagged with system information upon creation and altering and subject to any manipulation process that RDL offers. When turning from a node-local point of view on fact processing to a global viewpoint on the network, the transmission of a `context_fact` can be used for remote tasking. Upon reception, the rule engine of the node will add the fact to the fact repository and automatically behave as if the fact had been created locally.

This circumstance is especially advantageous in case stateful application behavior within a neighborhood of a node has to be achieved: Due to the broadcast nature of wireless communication, the coordination of physically close sensor nodes for applications such as object tracking or distributed event detection can be easily implemented by just sharing a specific configuration and/or control setting for involved sensor components with sending only a single `context_fact`. On the downside of dynamically tasking nodes in physical vicinity, the sudden appearance of multiple `context_facts` for one resource can lead to unexpected behavior. While having e.g. more than one `log_context` fact is not problematic (apart from eventual thrashing of the fact repository) and may well be intended, a variety of configurations for sensors received from neighboring nodes can entail race conditions. Also, the reception of `context_facts` can of course make sensor nodes vulnerable to denial-of-service (DoS) attacks. Countermeasures, e.g. authentication schemes to prevent these attacks have been explored intensively [113, 155, 116], and are therefore not further discussed here.

6.4 Quantitative measures of the FACTS runtime environment

Previous sections have discussed a variety of features, both in terms of hardware and software, that may be added to the vanilla implementation of the FACTS runtime environment on demand. This section provides a very brief, comprehensive overview of the actual impact of feature integration on both flash and RAM memory. Table 6.7 captures all data measurements that have been carried out in order to quantify FACTS-re deployment.

The ScatterWeb MSB430 firmware version 1.1 serves as the reference point with no user application being linked to it. In order to clarify the individual columns, one has to be aware that the text segment corresponds to the actual binaries, the data segment refers to the initialized data while the bss segment contains the uninitialized data specified within the program. Consequently, the sum of text and data segment will reflect the amount of flash memory occupied by the implementation, while the sum of data and bss segment provides a number for the statically utilized RAM memory.

A pure implementation of the FACTS runtime environment thus covers 16kbyte

Table 6.7: Memory consumption of FACTS-re implementation and various enhancements in comparison to pure ScatterWeb 1.1 firmware (in byte)

	text	data	bss	Flash	SRAM
Firmware	20100	24	1452	20124	1476
Firmware + SD card	27474	20	2494	27494	2514
FACTS-re	35992	24	2460	36016	2484
FACTS-re + ct	37068	24	2496	37092	2520
FACTS-re + sensors	42390	26	2466	42416	2492
FACTS-re + SD card	40768	26	2992	40794	3018

of flash memory and allocates roughly an additional 1kbyte of RAM. Note that data structures such as the fact repository (contributing with 930byte) and the event buffer are already included in this statically allocated memory, as no dynamic memory allocation besides the stack is supported. Due to recurrent, nested recursions during rule evaluation, a stack size of minimum 2kbyte has to be reserved to keep the rule engine operational and prevent system crashes.

The integration of compile-time information analysis to speed up the rule evaluation process is not costly: An additional queue for schedule acquisition in terms of RAM and 1kbyte flash for the corresponding software support have to be added. More demanding is the support of supplementary hardware, such as the SD-card or the optional SHT11 temperature and humidity sensor. Increased flash occupation is a result of linking the appropriate binaries of driver libraries and exporting read-/write access and sampling capabilities to the FACTS user, respectively. However, due to the block access mechanism, the SD-card requires to reserve a buffer page in RAM for caching the acquired values prior to eviction to flash. This justifies the tremendous increase of allocated RAM upon its integration into the firmware and FACTS, which is observable for both implementations.

The last question left unanswered is how many rules can be supported upon an embedded processor. Scanning the implemented rulesets, one can derive that 800 to 1000byte roughly correspond to ten rules, although sizes naturally fluctuate dependent on number of conditions, statements and diversity of implemented patterns for matching. With this rough estimate and an overall flash memory size of 55KB for the ScatterWeb platform, approximately 14kbyte to 19kbyte of flash may be utilized for rules, yielding up to 200 rules.

The great advantage in contrast to a native implementation is that the occupied RAM size will not vary and is thus independent of the number of rules deployed.

6.5 FACTS within Simulation Environments

Running real-world experiments with wireless sensor nodes can be alleviated with dedicated support and a good language abstraction. However, this does neither take the burden of having to cope with distributed application specification from a programmer nor shield her from the very time-consuming *develop-flash-debug* cycle inherent to WSN programming. Each sensor node has indeed to be flashed individually, configured according to application needs, distributed and the algorithm in question run in order to debug and test its performance, a circumstance that takes hours in case large network sizes are involved.

The common workaround is to resort to a simulation environment for testing instead. Certainly, reality can never be completely modeled due to simulation assumptions, statistical uncertainty may affect obtained results and one has to ensure that the chosen model reflects reality in the first place. But it is nevertheless a feasible approach to gain a general understanding of an algorithm's behavior and possibly explore its parameter space with controlled input values in a fast manner. Simulation environments are hence a valuable and valid tool within the development process to filter and address basic bugs of immature programs before turning to real-world deployments.

For testing and debugging purposes of RDL programs, two simulation variants are available: One can either rely on testing RDL programs with the help of the network simulator `ns-2`, which is especially favorable in case the focus is on exploring network behavior and performance of the chosen implementation, or one can utilize a command-line functional simulation tool which yields debugging at a very fine granularity. In both cases, RDL programs do not have to be ported or adjusted in any way, but simply the compilation target specified with dedicated compiler flags. In the following, only the basic ideas, design rationales and objectives are highlighted; the interested reader may refer to [153] for implementation details.

6.5.1 ScatterWeb on `ns-2`

The `ns-2` simulation framework is probably one of the most recognized discrete event simulators within the wireless networking domain. Implemented in C++, it provides a means to specify and run scripts written in the Tool Command Language (Tcl) denoting network and simulation parameterization. A wide variety of available radio-propagation models, implemented MAC and routing protocols as well as supported mobility models add to its popularity, making it an adequate choice for serving as a means to simulate ScatterWeb networks.

The approach to allow for the specification and execution of ScatterWeb software on top of `ns-2` is not restricted to or especially developed for FACTS, but rather targets to enable debugging and testing of any user application built on top of the ScatterWeb firmware. Hence, it is not an integral part of FACTS, but has been successfully used for several applications as will e.g. be presented in

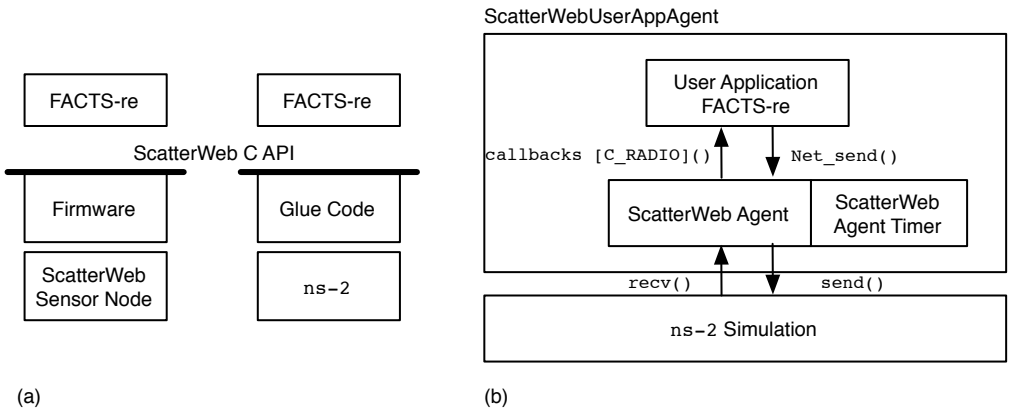


Figure 6.13: Running ScatterWeb and FACTS on `ns-2` (a) Conceptual model (b) Implementation

Section 7.4 later on.

Figure 6.13 (a) illustrates the conceptual model which has been adapted to achieve the envisioned integration: The key to avoid the need for adaptation is to provide a layer of glue code in between the user application. the FACTS runtime environment in this particular case, and the `ns-2` simulator. The interface of this glue code has to be identical to the one offered by the ScatterWeb firmware, thus both have to comply to the same API for transparent execution capabilities. Although this request is straight-forward, the actual implementation has to take several requirements into account:

- The ScatterWeb firmware is closely tied to its underlying hardware and implemented in C, while `ns-2` is primarily implemented in C++. Header files of the firmware have to be integrated due to the definition of data types and constants mandatory for application development. Name clashes upon linking ScatterWeb user application object code have to be considered.
- Static linking of ScatterWeb applications against the `ns-2` binaries with the help of implementing a simple compatibility layer is not possible. A network of sensor nodes has to be simulated, all of which have to maintain their own state. Static linking will however result in shared global variables among all nodes instead of allowing for replicated stateful information.

Since an automatic conversion of C to C++ failed to succeed, the approach towards integration has been the implementation of two C++ classes that cope with the above mentioned challenges, see also Figure 6.13 (b). The `ScatterWebAgent` class basically reimplements the ScatterWeb C API in C++ and deals with the interaction with the `ns-2` subsystem, whereas the `ScatterWebUserAppAgent` inherits this functionality and encapsules the application logic provided by the user

application, thus the FACTS-re to allow for its simulation. To bridge the gap to the `ns-2` network stack, the `ScatterWebAgent` is derived from its `Agent` superclass. The additional C++ class `ScatterWebAgentTimer` depicted in the Figure implements a means to enable event scheduling for the simulated firmware following ScatterWeb semantics.

Running RDL rules on `ns-2` is now a very easy task. The ruleset has to be compiled for usage in the simulation environment, which basically provides the image structured as discussed in Section 6.1 as a binary. A `ScatterWebUserAppAgent` has to be adjusted to include the FACTS runtime environment, meaning that global variables have to be declared separately, which then interprets the given image. To create a network of ScatterWeb FACTS nodes, simulation parameters such as network size and chosen MAC protocol have to be configured in a corresponding Tcl script for `ns-2`, and each node has to be instantiated to run the `ScatterWebUserAppAgent` code. Naturally, the Tcl script allows to furthermore specify calls to the functions available in that agent at discrete points in time, which can be used to simulate certain events happening or periodic sampling tasks.

All in all, this approach has proven to be very useful to understand network behavior prior to an actual deployment, but also to rerun real-world experiments with captured data traces to improve the overall quality of an implemented algorithm. The great advantage with respect to the usual approach of implementing an algorithm for usage in a simulator, and then re-implementing the same algorithm for actual devices, is that the complete code base can be kept. This not only saves a tremendous amount of time, but also ensures the fact that differences in performance are with high probability a result of real-world effects and not of different implementations. To support this hypothesis, experimental traces of the FenceMonitoring experiment have been re-evaluated within the simulation and were able to confirm this expectation, see also 7.4.

6.5.2 FACTS-hs: The Haskell backend for RDL rule evaluation

Functional languages such as Haskell comprise many advantages that make them quite appealing for software development: First of all, the ability to specify higher order functions allows for a very compact implementation of concerns while ensuring type safety, a circumstance especially viable in a state of prototyping as successive changes do not necessarily result in rampant source code. At the same time, program behavior can be understood more easily due to the inherent lack of side-effects upon function invocation. A nice feature of having a functional implementation of a system design is that it directly serves as a formal specification, enabling the assertion of important system properties if required.

The initial reason to resort to a Haskell implementation has been the quest to obtain a reasonable environment to experiment with RDL language design issues. In the end, it turned however out that the developed software serves for more than a mere prototypical implementation, but can be used to conveniently debug

complex rulesets at a very fine granularity simulating a distributed environment.

FACTS-hs, thus the functional backend for RDL interpretation, is implemented as a Haskell module, and combines the implementation of rule interpretation functionality as has been discussed intensively in Section 6.2 with its integration into a command-line simulation tool. Simply speaking, an arbitrary-sized network of sensor nodes can be created, all of which simultaneously execute RDL bytecode dependent on their individual context in terms of fact repository state and predefined events injected into the network at predefined simulation steps.

From an implementation point of view, a simulation is composed of its initial state and the mentioned list of events that occur while running, while a network is encoded as a list of nodes, each of which being represented as a tuple of its MAC address to serve for unique identification, its position within the network and a list of its rules, facts and available functions. In order to maintain simulation state, the simulation step count, the current time and the network of sensor nodes at that time as well as the queue of facts currently being transmitted, have to be available. Whenever an event is said to effect a certain node, the simulation step at which this event occurs has to be provided by the programmer. Then, a list of actions associated with the event in question is executed. Each update of a fact advances the simulation time by one to indicate a progression of time, overall allowing for 1000 updates per step. If during rule execution facts are scheduled to be sent, then these queued facts are to be virtually transmitted between communicating nodes by the simulation environment.

The public interface of the module provides constructors to allow for sensor node and network creation which will in turn run provided bytecode for simulation purposes. Furthermore, the FACTS-hs source offers constructors for rules and rulesets, conditions and statements, slots and fact entities. Unlike the bytecode image produced for the ScatterWeb backend, FACTS-hs bytecode is not a concise encoding of given runtime behavior of sensor nodes, but a simple syntactical transformation of the rules into Haskell sources that invoke the given constructors.

A simulation run iteratively transforms the current state of a given network with a given ruleset into subsequent states. Therefore, the consequences of event occurrences are evaluated at each node in the network and, in case rule conditions are validated, the corresponding actions executed at involved nodes to infer new network state. Since events manifest themselves as new facts appearing within a node's fact repository, this method can also be used to simulate sensor sampling or timer interrupts that request reactions. An application developer may step through the simulation to observe network behavior in response to given event sequences and analyze whether the deployed ruleset mirrors his expectation.

In contrast to the ScatterWeb on `ns-2` implementation, the Haskell backend has the advantage and at the same time the disadvantage that network behavior is *not* the focus of attention as lossless, instantaneous transmission is assumed. Hence, its prime application area is to aid a programmer in identifying causal dependencies between rules and rule parts respectively that may not be apparent at first sight.

6.6 Concluding Remarks

The emphasis of this chapter has been on the inner workings of the FACTS runtime environment, spanning from compilation over execution up to its integration into simulation frameworks. For each stage, we explained the basic design rationale and its implementation, highlighted possible and adequate optimization strategies and quantified the effects of chosen realizations. Furthermore, the design space for hardware abstraction has been extensively explored in order to provide comfortable access to underlying system functionality without loss of parameter control.

From a quantitative point of view, the results of implementing a rule-based middleware framework are very encouraging: Not only is the approach feasible from a perspective of mere implementation size and recursion depth for rule evaluation, thus practical on embedded hardware, it also nicely scales in terms of number of deployable rules, supportable runtime storage of facts *and* auxiliary hardware components for on-demand utilization. For an exhaustive, quantitative study, numbers on implementation sizes, especially of other runtime environments are not easy to obtain, either due to a lack of implementation or due to a lack of publication and/or free access to software. However, memory consumption for the vanilla implementation of the runtime environment (15.9kbyte flash, 1008byte SRAM (including fact repository)) is absolutely comparable to e.g. Maté (16kbyte flash, 849byte SRAM). Due to different design goals, chosen abstraction, underlying OS and implemented architecture one has to be aware that this can solely reflect the appropriateness, not the quality of the presented implementation.

Optimization strategies for decreasing bytecode size and speeding up execution time have proven to be sound (with respect to the optimization target) and efficient: The overhead incurred for tagging the bytecode with dependency information, as well as the additional cost for rule evaluation and scheduling at runtime are definitely legitimate in the face of 25% to 40% performance gain. Compile-time size reduction of the bytecode image comes at no cost at all since this is performed prior to the actual deployment on non-constraint machines.

Chapter 7

Utilizing FACTS: Implementing middleware and application-level functionality

Language pragmatics have already been discussed at a very abstract level in Chapter 5, revealing primarily *patterns* to capture data processing demands of wireless sensor networks and to circumvent general difficulties with event-centric programming. This chapter will finally turn to hands-on experiences, implementations and experiments carried out with RDL, exemplifying its utilization and the advantages of depending on a sandboxed runtime environment for sensor network tasking.

One explicit rationale for FACTS has been to *not* address the distribution dimension of abstraction by design, and therefore to *not* integrate a routing protocol into the framework per se. In general, it has been clearly a goal to avoid the implementation of any functionality whose support is not mandatory for all sensor network applications. Hence, supplementation instead of integration is the fundamental approach of choice. The first part of this chapter will therefore highlight how to supplement the framework with fruitful rulesets to incorporate middleware responsibilities, addressing distribution issues and coordinated behavior among sensor nodes in particular. Two different routing schemes, the very basic directed diffusion paradigm which yields data-centric communication with localized interaction, and a sophisticated routing protocol that optimizes multiple-source, multiple-sink routing with the help of feedback learning, have been implemented and evaluated to point out RDL's potential.

The second part of this chapter is concerned with the implementation of a real-world experiment for distributed event detection. Here, the focus will be on presenting especially the value of the tool chain, which enables to extrapolate network behavior from results derived in a medium-sized deployment by running the unchanged ruleset implementation in simulator.

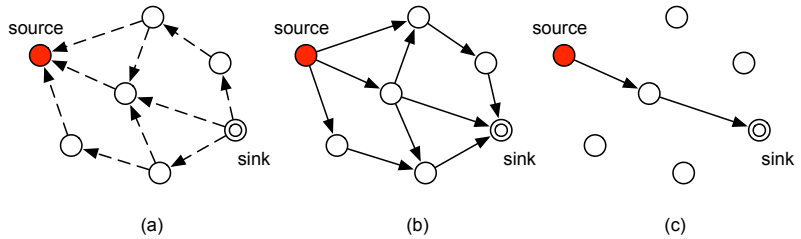


Figure 7.1: Directed Diffusion (one-phase pull) (a) Interest dissemination (b) Initail gradient setup (c) Data delivery along gradients

7.1 Routing I: Directed Diffusion

Given a deployment of sensor nodes within an environment to autonomously monitor a specific environmental condition or detect the occurrence of a predefined event, the question of what to do and where to store acquired (event) data is naturally a central concern. A prominent choice is to send this information to one or more dedicated nodes in the network (commonly referred to as *sinks*) which take either the role of gateways to other networks, posses higher capabilities in terms of hardware and/or possibly supply information on desired reactions. Numerous approaches have explored the design space of routing protocols to achieve energy-efficient [75, 143], adaptive [139] and scalable [50] information distribution within sensor networks, see also [8]. Amongst these, Directed Diffusion [83], already mentioned in the previous chapter, takes a prominent role due to its simple, yet versatile propagation scheme.

7.1.1 The directed diffusion routing protocol

Directed diffusion is a data-centric, multiple-source, single sink information dissemination protocol, which assumes all data generated by source nodes to be represented as named tuples of attribute-value pairs. To bootstrap information acquisition, a sink node describes its interest in application-specific data elements, which it then propagates through the sensor network, see also Figure 7.1 (a). In this first phase, the issued interests are forwarded within the network in a broadcast manner, and each participating node gathers information about its local, one-hop neighborhood. This knowledge is denoted in so called gradients, data structures that capture the next possible hop towards the direction of the sink, visualized in Figure 7.1 (b). The original version of the algorithm provides a step of path reinforcement afterwards. Therefore, source nodes infrequently send data along all available gradients, and the sink node reinforces a particular path that best suits the demands for delivery speed and path robustness. This information is then again reapplied by the sensor nodes in between source and sink to identify the single, best gradient to reach the target sink. Data delivery at full data rate

will be triggered in a last step of the original implementation.

The variant we chose to implement skips the reinforcement phase and directly sets the best gradient to the path with the smallest number of hops with respect to the sink, see Figure 7.1 (c), commonly referred to as one-phase pull directed diffusion. In the face of asymmetric links, this simplification may of course impact protocol reliability, but it also cuts the number of packets that have to traverse the network significantly. Route maintenance via periodic resends of sink announcements in terms of interests address the potential thread of broken routes instead.

For a better comparative study against the FROMS routing protocol [53] addressed in the next section, we furthermore extended the implementation to support multicast routing towards more than one sink. The key is to store one gradient per sink at each intermediate node, instead of one global gradient, applying straight-forward routing table semantics.

7.1.2 Implementation details

With its application-centric design, the representation of data in terms of tuples and its explicit reference to different routing strategies as rules, the paper describing the directed diffusion algorithm points out already the suitability of implementing it in a rule-based manner. While in the following, we will only discuss the most interesting rules for supplying an impression on language capabilities, the complete ruleset as deployed for the later on evaluation can be found the Appendix B.2.

The ruleset itself can be divided into different parts, each of which concerned with grasping a certain detail of the routing procedure. The first block is, as in most rulesets, dedicated to initialize global data, assign constant values and declare names and slots that are commonly used throughout the complete ruleset, see Listing B.2 (lines 1 - 38). Afterwards, protocol-specific parts follow: A set of rules that describes the effort of sink nodes for announcing `interest` in concrete data items (lines 39 - 99), a set of rules that captures node behavior upon reception of `interest` facts (lines 100 - 165), rules that describe how and when data is produced at a source node (lines 166 - 203), and finally a sequence of rules that implements data handling concerns (lines 204 -269) subsume node reactions to incoming facts.

This ruleset fragmentation already allows for two general conclusions; not all rules are functionally dependent on one another, which influences priority ordering constraints, and none of the nodes is actually in need of the complete ruleset for its proper operation. Recall that this second observation is an important part of the runtime optimization as these dependencies are exploited on a fact basis. The lack of functional dependency between all rule parts simplifies priority assignment: Not all rules of a ruleset have to be ordered appropriately, but rather the order within dependent sequences has to be watched carefully.

The question how exactly priority ordering impacts rule execution can nicely

be explained with the help of the interest handling rules (rules 5 - 9). A mandatory condition for all of these rules to trigger is the availability of an `interest` fact, the sufficient part chosen according to the specific reaction. The reaction, a possible definition or update of the gradient associated with the sink, is dependent on whether this particular `interest` is new, a duplicate with a better/worse hopcount towards the sink, an `interest` which serves for route maintenance or the handling node is the sink that issued the fact in the first place. Specification of these conditions can be greatly facilitated if the core triggering fact is removed after processing it, since then follow-up rules do no have to denote the complete search space in their left-hand-side. For instance, assigning a high priority to rule 5 which filters any `interest` that bounces from neighboring nodes at the sink, allows to neglect this possibility in all subsequent rules. Given a different order of the rules, the conditions will have to be customized to meet protocol semantics. All in all, the application of the *chain of filters* pattern described in Section 5.3.2 is directly visible in this sequence.

Looking at the ruleset as a whole, ordering of functional dependent sequences in respect to one another is arbitrary as long as statements do not affect the conditions of lower-priority rules, e.g. by removing, flushing, touching or setting a certain fact addressed afterwards. Data handling rules (rules 12 -18) and interest handling rules share no manipulation of the same facts, allowing to order these parts independent of one another.

From a protocol specification point of view, denoting directed diffusion in RDL is a straight-forward mapping of the original publication to source code. Data items such as `interests` and `gradients` are easily described as facts, with an `interest` carrying properties that denote its sequence number (`ann` = announcement number), the sink that issued the request for data (`sink`), the hopcount from the sink (`weight`) and the neighboring node that forwarded the fact (`neighbor`). This last information is actually not needed, as this could also be derived from the inherent `owner` property of the `interest` fact, but has been added for increased readability of the ruleset. Likewise, a gradient, which is similar to an entry into a routing table, comprises a property `sink`, indicating the final destination for data items, the neighbor that it has to be forwarded to (`neighbor`) for reaching this sink, the number of hops to reach that target node (`weight`) and finally the sequence number of the last route maintenance packet (`last_ann`). As can be directly derived from the source code, reaction provision in terms of rules is fairly easy: The complete protocol can be specified in solely 18 rules. The event-centric notion of protocol semantics are nicely mirrored in the RDL implementation as the required reactivity comes natural with a rule-based language abstraction and all data needed to encode protocol behavior is suitable for expression in terms of facts.

A quantitative evaluation of this and the following routing protocol will be provided in Section 7.3. Both share some similarities such as deployment topologies and metrics that may be well discussed together, and will be compared to a native implementation as well.

7.2 Routing II: Feedback Routing

The identification of the shortest path from a source to a sink, such as e.g. promoted in the above presented one-phase pull directed diffusion protocol, is the most common way to implement a routing algorithm. The FROMS routing protocol (Feedback Routing of Optimizing Multiple Sinks) optimizes for shortest path in the presence of multiple target sinks by application of a Q-learning approach and exploitation of the broadcast nature of wireless communication, providing a more elaborate way of information diffusion than directed diffusion.

7.2.1 The FROMS routing protocol

FROMS has been designed to cope with wireless sensor network deployments that require efficient data forwarding to multiple sinks at the same time. This circumstance arises for example whenever a node has to diffuse event notifications to more than one subscriber. The classical solution to this problem is the setup of a multicast tree, which usually involves prior knowledge about the network topology. Similar to multicast routing protocols, FROMS exploits the availability of shared paths as well. In the face of absence of a priori topology information, their presence has however to be learned from local interaction with neighboring nodes and hence cannot be utilized at bootstrapping time, but has to be gained during runtime instead.

Figure 7.2 illustrates this basic idea with the help of a very simple topology: Given a network that features two sinks and a source of data, these sinks will propagate their interest in specific data items via broadcasts to all nodes in the network, see 7.2 (a), which is similar to the interest dissemination that is used in the directed diffusion protocol. Gradient setup optimizing for the shortest path to both sinks will e.g. result in the network state visualized in Figure 7.2 (b). The shortest route to sink 1 as well as the shortest route to sink 2 will both have a length of 2 hops each, a typical configuration that will for instance be chosen when resorting to directed diffusion. However, while these paths resemble the local optima to individual sinks, the global optimum for this setup is depicted in Figure 7.2 (c). Here, an intermediate node that lies on the path from source to sink 1 *and* to sink 2 is used so that one hop may be spared. The key concern of FROMS is hence to identify, learn about and benefit from such shared paths in order to globally optimize information flow to multiple sinks, relying exclusively on local information.

To achieve this, and in contrast to directed diffusion, the protocol requests nodes to keep not only the single, best gradient towards a sink during an initialization phase, but to generate a full neighbor table with all available paths for later on optimization. Each data item sent out by a source specifies its target sinks, as well as the nodes that are in charge of forwarding this particular packet within the data payload. The protocol exploits the broadcast nature of the wireless medium by sending out data packets to the broadcast address, leaving it to

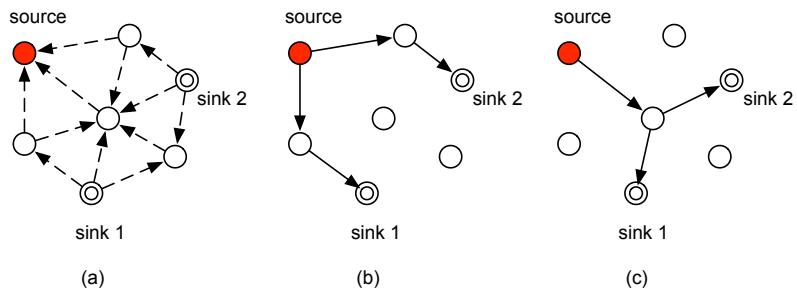


Figure 7.2: Idea of he FROMS routing protocol for routing to multiple sinks (a) Dissemination of sink announcements (b) Local optima for independent routing decisions (c) Global optimum

the nodes in network vicinity to decode forwarding responsibilities from the packet itself. Naturally, this will cut costs on the sending side, since each data item has to be issued only once for multiple recipients, however requires on the downside idle listing of all nodes at any time.

Upon reception of a data packet, a sensor node consults its neighbor table which denotes all possible options to route the data to the sinks the sensor node is requested to reach. Dependent on a given cost function, typically hopcount, these different options are evaluated and the best configuration is chosen. Correspondingly, those neighbors picked for forwarding are denoted in the data payload, before the packet is once again broadcasted.

Stand alone, a node will not be able to find globally optimal, shared paths. Although each node can be aware of the fact that it may reach several sink nodes using a dedicated neighbor from looking at the available gradients, any information that spatially goes beyond this first hop has to be provided externally. The presence of shared paths and their associated costs, have to be made available by neighboring nodes.

FROMS therefore relies on a Q-learning algorithm, responsible for the supply of feedback values to enable nodes to learn better, thus shorter, paths. Simply speaking, information on shared path availability is streamed in an upwards direction towards the source upon recognition. Implementationwise, this learning strategy is integrated into the protocol as a phase of exploration which a node can invoke to probe non-optimal paths. In a nutshell, a routing decision can follow two objectives, either to *exploit* the available information and choose the best path towards the given sinks, or to *explore* the network by choosing a random route and eventually discover shared paths it has not been aware of somewhere down along the path. In either case, the sensor node piggybacks the actual, best cost denoted for routing to the requested targets on the data packet to inform its predecessor about its local view on the network. Costs that were initially estimated to be higher can eventually be corrected in case this feedback information is overheard,

and shared paths identified via exploration can hence be made available.

In summary, FROMS comprises three distinct building blocks: a first phase in which sinks disseminate interests in data that are then used to setup initial cost estimates for routes, the implementation of the exploration/exploitation routing phase and a mechanism to issue and integrate feedback values appropriately.

7.2.2 Implementation details

Bootstrapping FROMS with the help of propagation of interests in data by the sink nodes is similar to the implementation of directed diffusion in rules, see Appendix B.3 for the complete ruleset. The differences here are simple naming concerns (`sink_announcement` instead of `interest`) and the usage of a context fact (line 47) for loop encapsulation, which is retracted after the initialization of the routing infrastructure. Data dissemination rules (lines 64 - 93) likewise differ only in minor details.

According to protocol semantics, the reception of `sink_announcements` will yield the creation of so called `PST_nodes` (path sharing tree nodes), which are basically representations of entries within a routing table. Each PST node fact therefore stores the neighbor from which the information has been received in combination with the sink that initially issued the request and the hopcount to reach it, denoted within the property named `cost`. Furthermore, a `unique_id` and a `flag` property are tagged to each PST node to facilitate identification and matching later on during processing. For each sink that a sensor node is aware of in the network, the minimal cost to route to this particular sink is stored in a separate fact `cachedMinCostSink`, which is used for restricting the number of PST nodes stored to only those not exceeding the overall path length by an additional hop (lines 95 - 168).

The protocol proceeds to the next phase as soon as the first data packet is received: The initial setup of the routing infrastructure is finalized by deriving shared paths from the available PST node information and simply denoting this knowledge in additional `PST_nodes`. PST nodes that target different sinks relying on the same next hop neighbor are merged into a new one, decreasing the cost to route to both sinks (which are combined using bitwise OR to encode their sharing ability). For instance, the availability of a PST node to sink 1 using neighbor 4 as the next hop, and of another PST node using neighbor 4 for to reach sink 2 will be merged into an additional PST node yielding sink 3 (thus both sinks), once again relying on node 4. Here, the elegance of pattern matching for information fusion is directly visible in the source code. After a rule needed to tag all PST nodes that fail to have a matching partner (lines 225 - 233), a single rule for iterating over all available PST nodes is sufficient, since it will be executed as often as there are still unmerged nodes in the fact repository. This is controlled simply by the `state_control` fact utilizing the *while-loop* pattern as described in Section 5.3.1. After successful setup, all needed information for routing is encoded and hence available for subsequent decision making.

A data fact received by a node can trigger different reactions, dependent on the role of the node itself and the actual payload of the packet. First of all, the information it conveys can be feedback on a prior routing decision, thus serve to propagate learned costs about formerly unknown shared paths. Rules for feedback handling (lines 293 - 349) are straight-forward. Each node temporarily remembers routing decisions by caching these in a fact named `feed_cache`. In case a data fact is overheard, the node checks whether it has such a fact from this particular neighbor on this particular routing choice and evaluates (a) whether the cost for this path (represented by the chosen PST node) and (b) whether the overall minimal cost for routing to the sink(s) have decreased. In either case, the corresponding facts are updated, in the latter all cached values for accelerated decisions on routing paths are also invalidated. Follow up rules will cleanup any cached feedback (lines 342 - 349) and bouncing facts (lines 352 - 357) to prevent repository thrashing.

A second possibility is that data is finally received by a sink node. Handling rules involve simple counting for calculation of the packet delivery rate (lines 366 - 385), issuing of feedback and removal of facts not requiring further routing (lines 386 - 407). Finally, a last option is that the data fact effectively denotes a request to forward it via the sensor node that is currently processing it. Then, the actual target(s) are extracted from the data packet, (lines 418 - 431) and the modus (exploration or exploitation) is determined (lines 437 - 458). We chose to set the percentage of exploration packets to 30%, leaving 70% of the packets for routing via the current best paths towards the sinks.

Implementing the exploitation strategy given solely the PST nodes requires a bit of effort since it resembles a depth-first traversal of a tree, with the target sink(s) being the root of this tree. All possible combinations to reach this root can be understood as subtrees that need to be checked for lowest cost to finally obtain the best path. Since this is a costly operation and sensor nodes are often required to route to the same (subset of) sinks, the first step is to check whether a cached route to the target sink(s) in question exists (denoted in the fact `broute`) that is valid and may be reused (lines 504 - 516). Otherwise, the algorithm proceeds with assigning a temporary minimum to the `broute` fact by searching for a PST node with minimal costs that feature the target sink(s) (lines 519 - 531). If the sensor node is responsible for forwarding to only one sink, processing will be stopped, the properties of the data fact changed, feedback cached accordingly and the data fact finally sent out (lines 533 - 562). However, in case multiple sinks are to be reached, the complete imaginary tree has to be traversed. Likewise, the follow-up rules implement such a search, matching all possible paths, storing these within temporary `croute` facts (lines 626 - 634), which are compared to the current best route (lines 637 - 657). In the end, this fact finally bears the required information and is used to adjust data fact properties before actual transmission. As has been stated above, this `broute` fact will also serve as a cache for future routing decisions as long as it is not invalidated by incoming feedback or requests for different routes.

With respect to exploitation, denoting the exploration strategy is again very

easy: Given a random value, the sensor node simply uses this to implement a function mapping it to the available PST nodes for random choice (lines 766 - 768). Eventually, a matching node is furthermore obtained, or in case this mechanism failed due to very sparse number of PST nodes, a match-any strategy as a fallback mechanism is implemented.

Overall, denoting FROMS with the help of RDL rules is a bit more complex than the implementation of directed diffusion for two reasons: First of all, the protocol itself features more options for nodes to decide on data handling, which naturally increases protocol complexity accordingly. While this is however true for implementations in any language, the second can be directly owed to the promoted data model in RDL, representing any available data in terms of facts, lacking higher-level data structures directly available to the programmer. As a consequence, more effort has to be put into implementing things as e.g. trees or iterators, which is mirrored in the implementation for the exploitation strategy. However, the strict separation into individual phases once again allows to fragment the complete ruleset featuring 51 rules into functional chunks of way below ten interacting rules, facilitating the implementation drastically.

7.3 Comparison of native and rule-based routing protocol implementations

In order to evaluate a rule-based against a native implementation, we will rely on a two-step process: In a first step, we investigate general performance metrics of the routing algorithms implemented in both a native and a rule-based manner. If obtained values are similar (or within the bounds of acceptable deviation), this will ensure that a direct comparison is valid in the first place, as it is likely that we face semantically similar implementations of the same algorithm. Afterwards, the overhead of rule-based interpretation instead of native execution will be discussed. We will resort especially to the second routing ruleset for doing so, as this features more complex processing requests which are able to better illustrate the impact.

The native implementation of both, directed diffusion and FROMS presented in the next section, have been provided by Anna Förster, with results publicly available in [54].

7.3.1 General testbed setup and protocol parameterization

The results discussed in the following have been obtained running directed diffusion (DD) and FROMS on the testbed depicted in Figure 7.3, using ScatterWeb MSB430 sensor nodes flashed with the ScatterWeb firmware version 1.1. The sink nodes (node 1 and 2) are placed directly opposite the source node (node 6), with the visualized multi-hop topology artificially enforced using a packet filtering script on each sensor node. This step has been mandatory to ensure comparability of all test runs.

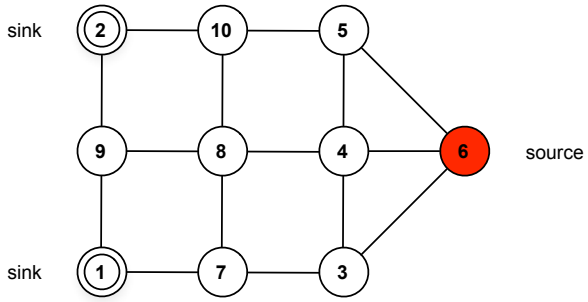


Figure 7.3: Testbed topology for evaluating directed diffusion and FROMS

For each run, both in DD and FROMS, the source node issues 100 data packets to be routed to both sinks. During the protocol bootstrapping phase, the `interests` and `sink_announcements` respectively, are diffused every 15 seconds until the first data packet is received. Then, directed diffusion will issue route maintenance packets every 50 seconds, while FROMS resorts to the exploration and exploitation scheme as described above. All results displayed in the following are averages from three consecutive runs of each algorithmic implementation.

7.3.2 Evaluation of routing protocol characteristics and performance

Figure 7.4 summarizes measured values in terms of packet delivery rate and associated costs for routing for each protocol, comparing native and rule-based implementation denoted on the x-axis of the diagram. Since the overall impression of fairly similar value ranges can be confirmed for similar protocols, the first and foremost conclusion drawn from these experiments without looking into the details is that, although different languages have been used for protocol implementation (C for the native, RDL for the rule-based variants), protocol semantics are captured equally well. This is a mandatory prerequisite for any further investigation.

For an evaluation of protocol performance, we rely on three distinct metrics, the well-known packet delivery rate (PDR) which resembles the ratio of received packets by the sink nodes divided by the number of packets initially issued by the source, and two metrics that express how many hops a packet has traversed within the network in order to reach its destination in relationship to the issue rate and the success rate, respectively. The cost per generated packet is hence the ratio of overall traffic load on the nodes, thus the sum of all data packets sent within the network during an experimental run, and the number of issued packes, while the costs per received packet calculates this in relationship to the delivery rate. Both ratios are displayed in terms of number of transmissions (ETX) each implementation features in average. Given the topology that is illustrated in

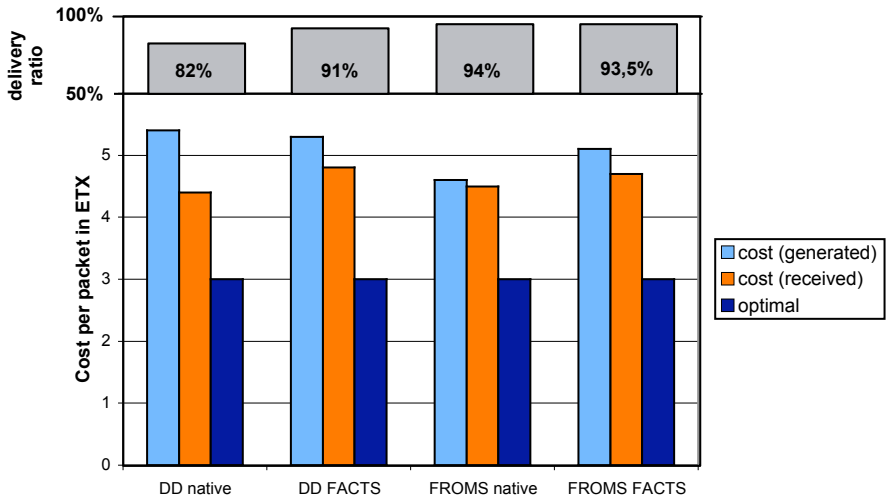


Figure 7.4: Evaluation of routing costs and delivery ratio of DD and FROMS in different implementations

Figure 7.3, the optimal value, resembling the optimal path from the source to the sinks is three, as these hops are necessary to reach the sinks under the premise of broadcast communication.

The value showing the most deviation for the different implementations is the packet delivery ratio for directed diffusion. While an implementation running on the FACTS framework yields a rate of 91%, the native implementation is only able to confirm a rate of 82% successfully delivered packets. The reason for this difference can probably be attested to the rather unstable protocol itself: Timing of route maintenance packets has a tremendous influence on subsequent data transmissions as coincidently issued data and interest packets lead to a peak load, and result in collisions not only corrupting data packets targeting sink nodes but also maintenance packets. Formerly stable routes can then be replaced with longer paths over less reliable links, which negatively impacts the delivery rate. A rather high variance for the PDR is already observable in individual test runs of directed diffusion. In contrast to this, the PDR of FROMS is almost the same for both implementation, with a PDR of 94% for the native and 93.5% for the rule-based variant. Lost packets are in this case due to usual collisions and interference, but no protocol-inherent reasons can be derived.

The cost metrics to evaluate path lengths of routing decisions visualize for the directed diffusion protocol that either the FACTS implementation has a tendency towards resulting in slightly longer routes than the native one, or that the increased packet loss for the latter is especially observable at early hops on the paths to the sinks. Since the costs per generated data are almost equally high for both and the

Table 7.1: Node-local memory characteristics of native and rule-based routing protocol implementations

	DD native	DD FACTS	FROMS native	FROMS FACTS
ROM usage	8396 byte	1682 byte	12358 byte	5660 byte
RAM usage	2932 byte	max. 200 byte	3326 byte	max. 656 byte

native implementation features a smaller cost value for received packets despite the fact of a lower delivery rate, either situation is a valid explanation.

For the FROMS routing protocol, both implementations nicely illustrate the ability of FROMS to decrease the path length for reaching multiple sinks via its learning strategy. Here, the native implementation clearly outperforms FACTS, a circumstance that is at least partially a result of a node failure during experimentation, which the protocol however bypassed choosing a longer route. Also, the question how fast good, thus shared paths are chosen for exploration impacts protocol convergence time, and as a result the overall number of packets to be routed along the best path. From the traces we were able to derive that the chosen exploration strategy may not have been optimal for the RDL routing implementation, since path convergence has been obtained rather late during the experiments, testing a lot of non-optimal routes previously.

Once again, it is noteworthy to mention that the intention of this evaluation is neither to provide an in-depth analysis of different routing protocols, nor to attest RDL any superiority in regard to performance metrics over a native implementation. Rather, this section provides the essential background information for confirming similar, if not identical protocol semantics regardless of the chosen language for implementation.

7.3.3 Evaluation of node-local characteristics of distinct protocol implementations

From the perspective of node-local abstraction provision, it is much more interesting to evaluate characteristic metrics for analyzing the quantitative value of the FACTS framework in regard to memory consumption and overhead in terms of processing time. Table 7.1 provides numbers on the first issue, while the latter is displayed for individual routing decisions by the nodes in Figure 7.5, respectively. While often times, the value of abstraction is measured in lines of code (LOC), this metric is to our understanding too vague for truly pointing out differences in implementations as soon as one turns from toy to real application, as the experience of the programmer has too big of an influence on the actual outcome of the results. As we will discuss in the following, already the numbers provided below have to be put into context in order to prevent a misinterpretation of the results.

Concerning memory consumption in terms of the binary image of the implementations (ROM), the resulting values can once again verify that one of the design goals for FACTS, the provision of lean and concise bytecode for sensor network tasking, has been met. The image for directed diffusion accounts for only a fifth of the size of the native implementation, while the FROMS image in bytecode takes up roughly 45% of the ROM compared to the native implementation. Clearly, common tasks such as packet assembly and disassembly, event dispatching and queueing are responsibilities that within FACTS are shifted from the programmer's to the runtime's responsibility and can therefore be spared during protocol implementation. Hence, one has to be aware that mere protocol semantics (FACTS) and protocol semantics in combination with pre- and postprocessing concerns are mirrored in these values. Nevertheless, this also illustrates that the FACTS approach is able to efficiently shield a programmer from commonalities in WSN tasking, allowing her to focus on the actual problem specification.

A comparison of the numbers denoted for RAM usage has to be correlated to the actual implementation strategy for denoting the routing infrastructure in order to grant a just analysis. The rule-based approach here has been to only keep absolutely relevant facts, thus knowledge acquired for routing decisions that is not volatile, in its fact repository. To put it the other way around, all data that is subject to frequent change is rather recalculated on the fly than stored, a circumstance that will impact processing time as will be seen in the following. In contrast, the native implementation favors to keep all information once derived available in RAM to increase protocol reactivity.

As a consequence, RAM allocation for directed diffusion is very limited with FACTS: solely two gradients (one for each sink), the `system` fact for initial protocol setup and a possible `interest` or `data` fact arriving at the node coincidently at the same time have to be materialized in the fact repository, which in turn will occupy 200 byte of RAM. In contrast, the implementation of the directed diffusion protocol allocates a huge part of the RAM (almost 3 kbyte) for route encoding, which is partially owed to the fact that this part of the implementation is shared by both, directed diffusion and FROMS for reasons of decreasing development time.

Obtained values for the FROMS implementation substantiate this observation. All full routes that may possibly be derived from the PST nodes given after the announcement phase are pre-calculated by the native implementation and stored in RAM, which in turn accounts for the very high RAM utilization of the protocol implementation. Facing the encoding scheme for facts, which tags individual tuples with additional system related parameters for event correlation, we estimated the costs in terms of memory consumption to gain insights on whether this approach would be a feasible choice for an RDL implementation, too. Given k sinks in the network and n neighbors (indicating the density of the deployment), the worst case memory consumption M_{PST} for the PST nodes is denoted in Equation 7.1.

$$M_{PST} = n * (2^k - 1) \tag{7.1}$$

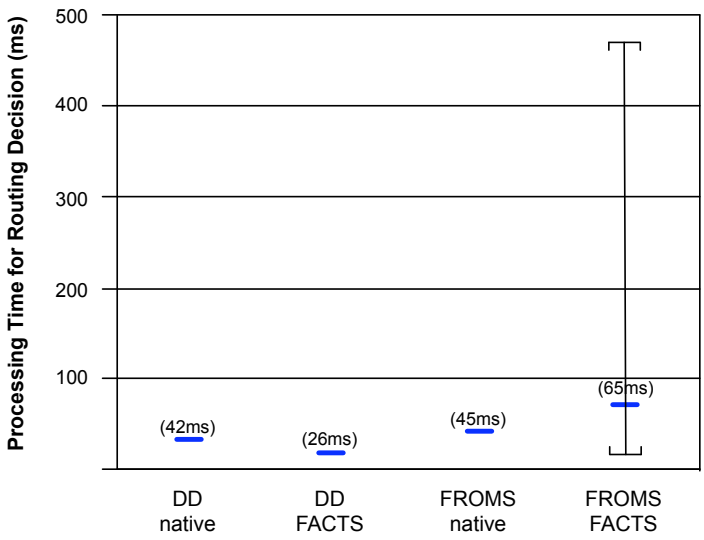


Figure 7.5: Processing time for finding a route to the sink(s) in DD and FROMS for different implementations

When implementing FROMS on the sensor nodes, this memory has to be at least available in order to encode the routing table information unless the tree is pruned using heuristics. With an encoding size of 46 bytes per PST node in the current implementation of FROMS in RDL rules, a maximum density of four neighboring nodes and two sinks, 12 PST nodes have to be materialized in the worst case. However, all paths in a backwards direction towards the source are automatically pruned due to the threshold definition for storing PST nodes, so that a number of 8 PST nodes (386 byte) can be denoted as the actual worst case for this implementation/topology combination.

If we now want to additionally keep all possible routes to the sinks, the imposed cost of $C_{fullroutes}$ can be calculated as follows:

$$C_{fullroutes} = n^k + \sum_{i=0}^{k-2} \binom{k}{i} * n^{i+1} \tag{7.2}$$

Since each route in FACTS consumes 40 byte, the approach of storing all available routes would be manageable for the given topology, yet render the routing protocol to be the sole instance that is able to be deployed on the sensor node. From a software development point of view, this is to our understanding not a reasonable choice a programmer would make. Therefore, we accept the difference in the individual implementations and instead evaluate the impact that this will have on runtime performance.

Figure 7.5 gives an overview of the average time needed for a single routing decision derived from all nodes of the network in all four implementations. In the

light of knowledge of the encoding scheme of routing paths in both of the native implementations and that all routing decisions either result in a single lookup (in case of directed diffusion and route exploitation in FROMS) or the creation of a random value prior to a lookup (FROMS exploration), the depicted results of an average processing time of 45ms time are not astonishing. The rule-based implementation of directed diffusion yields with an average value of 26ms roughly the same value range, which is slightly lower due to the missing packet disassembly and assembly steps in processing.

The average time to find a route in the rule-based implementation of FROMS is with 65ms very promising at first sight, indicating that the overhead of bytecode interpretation and subsequent runs of the rule engine is absolutely viable. While this is true for most of the routing decisions, excellent protocol reactivity can however not be attested in all cases: In order to allow for a fair judgement, the absolute dispersion of values, depicted as well in the diagram in Figure 7.5, has to be pointed out, which reveals the downside of choosing to optimize for storage over computation time for routing table setup. As soon as a re-computation of the best available route from all PST nodes becomes essential, this may take up to 475ms in the worst case for nodes that are very well connected. Since the native implementation may resort to complete routes stored in RAM (which is not feasible for the RDL implementation), this computational overhead is avoided. Protocol reactivity is thus subject to a high variance, especially during the initial phase which features frequent updates in route estimations. A straight-forward solution to overcome this potential threat when utilizing FROMS in real-world applications is to schedule route re-computation explicitly in idle times, resorting to potentially non-optimal paths when fast routing decisions are needed. This filtering of local minima of PST node entries can instead be implemented in a single rule, rendering the costly depth-first tree traversal for searching obsolete.

7.4 Fence Monitoring

FACTS has not only been used to implement supportive measures for applications, but also to test and run applications themselves. The experimental use case briefly introduced in this section has been carried out to evaluate the benefit of collaborative, in-network event processing. Although the availability of localized data processing capabilities is widely regarded as a key feature of wireless sensor networks, most real-world deployments still rely on data-agnostic reporting schemes, a circumstance we wanted to address.

The use case we decided to implement has been concerned with physical intrusion detection, evolving around the question whether it is feasible to distinguish a person climbing over a construction fence from other movements the fence may be exposed to. In order to emphasize the programming perspective, the focus here will be on the actual implementation of the proposed algorithm and on the utility of the available simulation tools, rather than on a general overview of experimental

Figure 7.6: Deployment of the a ten-piece construction fence in the patio of the institute (a) Schematic illustration (b) Picture of the actual deployment (c) Sensor node mounted to the fence

concerns. A complete discussion of related issues such as e.g. sensor calibration or deployment challenges can however be found in [154].

For running our fence monitoring experiments, we installed a ten-piece construction fence in the patio of our institute and mounted one ScatterWeb MSB430 sensor node to each element as depicted in Figure 7.6. Furthermore, we defined six distinct events that a fence may face, i.e. being kicked, being shaken for a long or a short period of time, being used for leaning against it or peeking over it to see beyond, and finally crossing it by climbing over the fence. With the help of sampling the on-board accelerometer, the task has been to issue an alarm as soon as a person crossing the fence has been identified from the sampling pattern. Naturally, an analysis of the different events to find bounds for filtering background noise and a pattern to match the climb event has preceded the actual detection runs for calibrating the event detection architecture.

7.4.1 Implementation details

The actual event detection algorithm conforms to a hierarchical data processing architecture, a valuable pattern for processing as already pointed out in Section 5.3. The algorithm itself therefore proceeds in subsequent filtering steps, first removing background noise, then proceeding to local event detection and, in case of success, justifying or revoking the potentially detected event with the help of a neighborhood query. Only when all stages have been passed by an event candidate, the acquired data samples are classified to represent an intrusion and hence an alarm is being issued. Based on a simple spanning tree, this alarm is forwarded to a predefined sink node.

Listing 7.1: The detection rules within the Fence Monitoring ruleset

```
1
2  rule aggregateShakeEvents 190
3  <- exists {shake}
4  <- eval ((count {shake}) >= localAggregationMinShakeEventsTrigger
       )
5  <- eval ((sum {shake duration}) >=
       localAggregationMinCombinedShakeDuration)
6  <- eval ((sum {shake duration}) <=
       localAggregationMaxCombinedShakeDuration)
7  -> define climb [confidence = ((max {shake intensity}) * (max {
       shake duration}))]
8  -> retract {shake}
9
10  [...]
11
12 rule evalAcksOnTimeout 110
13 <- exists {neighborhoodAggregationTimerExpired}
14 <- eval ((count {ack}) >= neighborhoodAggregationMinAckTrigger)
15 -> define alert [confidence = {climb confidence}]
```

The complete ruleset implementing the fence monitoring application can be found in Appendix B.4 and relies on the provision of basic events, so called `shake` facts down on the lowest layer by the firmware. Data samples are simply aggregated into `shake` facts as soon as the intensity of the sampled acceleration crosses a threshold.[1] Each fact is tagged with two properties, the average measured `intensity` and the overall `duration` of this event before being pushed into the FACTS runtime for processing.

From here on, the ruleset takes over the processing routine, see also Listing 7.1 which denotes the two rules for local, as well as neighborhood event detection. A local event, thus a `climb` fact is created, whenever the number of available `shake` facts exceeds a threshold (we derived a number of three shakes to serve as a good estimate from the test runs), and the sum of their durations is within given bounds (lines 2-6). Another rule, not displayed here, is in charge of purging `shake` facts after a short period of time to react and process only fresh data.

In case a potential `climb` event has been detected, this hypothesis is tested by broadcasting the fact to the one-hop neighborhood of nodes for verification after a short delay. Network proximity exceeds spatial proximity in this scenario, so that physical neighbors will by all means be reached. Each node receiving a `climb` fact will check its fact repository whether it can confirm the observation, and either acknowledge or reject the raised claim by sending an `ack` or a `nack` fact accordingly. Upon reception of at least one acknowledgment, neighborhood event detection is triggered, and an `alert` sent to a sink (lines 12-15).

[1] The intensity is a measure we defined as the absolute value of the sum of the differences of current and previous sample in each dimension.

The small scale of deployment had the drawback that practically all nodes were reachable within one hop, even after lowering the transmission range by setting the signal strength of the transceiver to the minimum. To measure the impact of in-network event detection and its implicit compression via events upon the network load, we therefore re-applied the traces within the `ns-2` simulator, see also Section 6.5, and crafted a bigger, more realistic deployment scenario.

7.4.2 Results and outlook

The results obtained for detection have been quite reasonable, especially with respect to other deployments such as [149]. For evaluating the actual quality of event detection, we utilized two statistical metrics for binary classification, namely *sensitivity* as a measure of correctly identified intrusion events, and *specificity* as a measure of the share in false positives. We then differentiated between local and neighborhood event detection, and measured the impact of the additional step of in-network filtering.

Overall, the algorithm performs well in terms of not rejecting true positives in the local event processing stage: With a 100% value for sensitivity, no intrusion event has been discarded early. On the downside, the specificity value of 41.3% indicates that the number of wrongly classified events is still high, since almost 60% of the event candidates do not represent climbs. The neighborhood filtering is able to improve this specificity by 12.0%, unfortunately at the expense of a 13.3% decrease in sensitivity.

Looking at the networking aspect of in-network event detection, early filtering clearly has the anticipated effect of significant reduction of network load, even though additional traffic is incurred by the neighborhood event detection. We utilized a simulation scenario that featured a virtual fence around the US Embassy in Berlin, in combination with a setting of 10m transmission range for the sensor nodes to derive concrete values. Naturally, raw data streaming (thus streaming of basic events i.e. `shake` facts) turned out to considerably stress network capabilities. While the transmission of event candidates (i.e. `climb` facts) is able to reduce this traffic by 79.3%, the overall reduction can be further improved to 93.4% when relying on the complete event filtering hierarchy.

The fence monitoring deployment has been a great scenario to implement, deploy, test and evaluate a small scale, real-world use case with available components in terms of hard- and software. A tremendous amount of work has since been dedicated to improve especially the event detection algorithm and supply more sophisticated algorithms for pattern acquisition and classification, see [48, 146] for further reading. From a networking perspective however, the impact of information aggregation in different granularities and topologies has been able to be quantified even with given prototypical implementation, see [154] for an in-depth quantitative evaluation.

7.5 Concluding Remarks

The main theme of this chapter has been to analyze whether real-world, wireless sensor network specific problems can effectively be solved utilizing the FACTS middleware framework. Therefore, a comparative study of two, in terms of complexity very distinct routing protocols, implemented in a native and a rule-based manner, as well as the implementation of an exemplary use case try to provide valuable insights.

From the first part of this chapter, two claims concerning quantitative metrics can be directly confirmed: RDL bytecode is in general very concise with respect to native implementations of equal protocol semantics. Furthermore, in the average case, the overhead for bytecode interpretation via production rule scheduling in terms of actual processing time, is for the implemented problem at most approximately 45% higher than for native scheduling, given however different implementation strategies. Naturally, one has to be careful to not misinterpret this number: Rather than serving as a fixed value for the impact of interpretation, it is only able to provide a rough estimate for a general indication. Then again, the different programming paradigms will always yield application- and paradigm specific implementations, so that this at first sight vague conclusion is certainly more to the point then running a benchmark program for deriving a coherent valuation.

On the downside, the implementation of the FROMS routing protocol has also been able to push the framework to the limits. Especially the definition of complex data structures allowing for very fast and efficient access is a problem that cannot be sufficiently handled with the chosen data model and pattern matching strategy. The unconditional support of spatio-temporal event processing patterns via inherent tagging of facts furthermore imposes an inevitable overhead which is counterproductive in case large amounts of data have to be encoded very efficiently. The specification of index structures to speed up the access to specific data items can however be implemented in an application-specific manner, but is not directly supported by the language itself.

But not only quantitative values of the FACTS framework have become apparent in this chapter. Once again, all three implementations presented above revealed the advantage of empowering a developer with the possibility to adopt a clearly problem-oriented viewpoint for protocol or application specification. Defined data is, with the exception of a few temporary facts that aid in matching and filtering in the FROMS implementation, purely application- or protocol-specific. Especially the fence monitoring ruleset is able to convey this circumstance with the encapsulation of different event types in different facts after successful recognition. Meaning is directly encoded within the fact specification, and reactions to particular fact instances derivable from individual rules. All in all, this renders rule-based protocol implementation with RDL on top of FACTS not only suitable, but very valuable from a programming perspective.

Chapter 8

Conclusions

This thesis has been dedicated to explore the design space for abstraction provision in wireless sensor networks. Autonomous, small devices, deployable physically close to phenomena of interest, reprogrammable and even enabled to interact with other devices and networks bear a tremendous flexibility to serve as a technological basis for new, exciting applications. Unfortunately, the great potential of these networks has up to date not really been exploited, a fact that can be attributed to a number of unsolved problems, including operational and economical reasons. Costs are for instance still high and prototypical implementations beyond pure scientific work rare, which makes investments in this technology risky. Benefit and utility with respect to established technologies is yet to be substantiated.

A key aspect that hinders fast technological adoption is an observable lack of intuitive handling of wireless sensor networks and their individual nodes, respectively. Effective programming for this domain is highly demanding as network *and* device-level challenges are most of the times directly visible to the application developer, thus have to be thoroughly understood and objected accordingly. Unlike for instance the enormous amount of creativity that the relatively simple development of web content spawned early this century, this domain requires sophisticated software development skills which increases the barrier for getting in touch with the technology in the first place.

As a result of an in-depth analysis of the domain itself, of its envisioned applications, its preferred software architectures and proposed middleware solutions, the primary challenges and drawbacks that have to be addressed to overcome this situation have been pinpointed at the beginning of this work. The main contribution of this thesis is the provision of a holistic node-level programming model and middleware framework to facilitate wireless sensor network tasking. The framework itself comprises a domain-specific language that enables the specification of nodal behavior with the help of declarative rules, a corresponding runtime environment, implemented to run on top of both real sensor hardware and within simulations which ensures robust application execution, and a rich set of middleware abstractions to further complement the approach.

8.1 Contributions

The FACTS framework objects the described problem of low productivity due to the high domain-specific demands and the tedious software development utilizing two well-known abstraction paradigms, fusing a domain-specific language and supportive measures into one coherent programming model for wireless sensor network tasking.

8.1.1 Provision of a classification model for wireless sensor network abstractions

Abstraction from details, simplification of non-trivial processing and data concerns by means of shifting the responsibility for their objection to a dedicated piece of hard- or software is a fundamental mechanism to deal with complex problems. Since software development for the wireless sensor networking domain can well be categorized to bear such structural complexity, a number of approaches that provide strategies and support to overcome the challenges of this particular domain have been proposed in the past.

This thesis not only presents current state of the art and summarizes different abstraction flavors, individual advantages, shortcomings and overall findings, but furthermore provides a *general* classification scheme to grasp the inherent conceptual ideas of a certain approach, as well as to conceive its functional emphasis. The three main dimensions of abstraction that can be utilized in a WSN context, namely abstraction from distribution and networking concerns, abstraction from programming perspective discrepancy as well as abstraction from intrinsic software organization problems, are valuable instruments to reveal the inner workings of any past, present and future abstraction implementation for this area. An additional application of a set of classical metrics provided in this thesis to judge approaches from a functional viewpoint will complement the conceptual evaluation. Overall, this enables a very fine-grained classification of solutions, which may easily be utilized when searching for a suitable abstraction addressing a dedicated implementation problem.

8.1.2 Specification of a domain-specific programming language for wireless sensor networks

To our understanding, the foremost challenge that has to be met for bridging the gap towards improved sensor network tasking capabilities is to ensure the availability of a simple, yet versatile means for problem-oriented task specification on the one hand, and to warrant robust software development on the other. Due to human nature, language and problem conception are tightly coupled to one another, rendering the provision of an adequate language substrate in order to increase expressiveness a great tool to handle difficulties that arise from target and problem domain antagonisms.

A major contribution of this thesis is the definition, implementation, application, extension and evaluation of a concise, powerful programming language especially designed for the wireless sensor networking domain. Accepting the primarily event-driven view upon software development instead of artificially disguising it, a high-level, rule-based programming language called RDL (ruleset definition language) has been specified to facilitate sensor node tasking. Interleaving general concepts of ECA rules and production rule semantics not only allows for the specification of event-based reactivity of a sensor node, but also to derive knowledge, e.g. on its current context or processing state from its available information. Precise execution semantics of RDL rules are denoted in a formal manner in order to enable implementation-independent language perception.

8.1.3 Implementation of a holistic, node-level programming framework

From a practical perspective, the provision of an abstraction is solely valuable in case it simply works. Evidently, this demand not only concerns its mere availability and adaptability to the encountered problem, but also subsumes the request for reliability, general ease-of-use and support in all phases of the development process.

The core of this thesis evolves around the provision of a holistic, node-level programming framework that enables wireless sensor nodes to interpret rule-based programs denoted in RDL. Intrinsic challenges such as event ordering concerns, direct exposure to hardware interfaces, packet handling and the obligatory need for manual stack management on embedded devices are shielded from a programmer and supplied via easy-to-use interfaces. The stack itself is for instance neither visible nor manipulable by a developer, shifting any responsibility for proper memory handling to the FACTS runtime environment, which in turn warrants robust rule execution at runtime.

A variety of tools, e.g. for network simulation and debugging, and additions, e.g. the provision of specialized facts to encapsulate configuration and controlling concerns, to complement the programming framework have been proposed, tested and their capabilities explored. A programming framework for a domain-specific language exhibiting an equal state of maturity is to the best of our knowledge up to now not available for the wireless sensor networking domain.

8.1.4 Exploration and exploitation of the design space for optimizing the proposed programming model

Regardless of the target platform or deployment area, an optimal abstraction is lightweight, highly flexible, easy-to-use, conceptually coherent and provides powerful support for a dedicated problem area. Naturally, it has been a primary concern of the framework proposed in this thesis to promote as many of these design goals as possible without breaking the inner, conceptual model of the FACTS approach. Therefore, optimization strategies to obtain lean, modular bytecode

to support flexible sensor node retasking have equally been explored as well as mechanisms to speed up the overall processing time of rule interpretation. A thorough quantitative evaluation of their impact, relying on different use cases and implementations allows to draw a positive conclusion on the utility of available enhancements.

Pushing an approach to its limits often reveals interesting facts about the advantages and bounds of the proposed abstraction. In order to truly test the FACTS programming model and grant a fair judgement with respect to prevalent imperative implementations in this domain, a comparative study has been carried out, utilizing two established routing protocols with highly distinct demands in terms of protocol complexity. From a qualitative point of view, the RDL language is clearly superior when it comes to denote causal dependencies, reactivity and event patterns, runs however short in case very efficient encoding and access schemes are needed for vast amounts of data. Correspondingly, highly complex search operations within the data space impose a measurable overhead on the sensor node due to subsequent rule scheduling. However, one has to be aware that this is not the average case as we were able to confirm, but rather a finding revealed when deliberately searching for model bounds. Being a high-level language abstraction that depends on bytecode interpretation on embedded hardware, FACTS indeed performs way better than ever imagined.

8.2 Outlook and Future Work

The FACTS programming model itself is a self-contained approach that nicely fulfills its initial goal. Three aspects would however be interesting to further investigate, one concerning the language, another probing additional supportive measures and a final to unleash the power of rule-based interaction within a broader context.

A next step to take towards more reliable language design is to introduce type safety within the RDL language syntax, and naturally to ensure this by appropriately customizing the compiler. Stand-alone, this is solely an implementation issue rather than being a research question. One drawback that became apparent within the FROMS implementation has been the indifferent usage of facts to denote all data in the system, a problem that in databases, which equally depend on a declarative data model, has been tackled with the introduction of binary large objects (BLOBs). A very promising thing to look into is whether there exists a deliberate data structure apart from facts that seamlessly integrates into the pattern matching model, however provides a means for efficient storage capabilities that software development for embedded devices can benefit from in situations which lack explicit event semantics.

Putting differential sensor network retasking into practice is a second area worth investigating. The availability of simple means for over-the-air substitution of rulesets naturally gives raise to a number of curiosity-driven experiments yielding self-aware, autonomous adaptations of sensor nodes at runtime and spon-

taneous exchange of rule configurations. Especially in a mobile environment that features distinct, spatial surroundings and eventually calls for context-aware node behavior, flexible rulebases seem to be a sensitive means to support small, low-capacity devices yet enable powerful, in-situ reactivity.

A third area that probably has the capability to become a killer application for rule-based interaction is when semantic web technology really hits the embedded domain. Although this area is not new and a lot of work has already been contributed on the integration of smart objects into the real world, an increase in maturity, opening up the development to non-professionals by means of dedicated, but not too constraint programming models and cheap technology can eventually trigger exciting developments within the next years not foreseen now.

References

[1] ILog - An IBM Company for Business Rule Management Systems. Company website avaiable at: http://www.ilog.com/. Last access: 17.3.2009.

[2] SunSpotWorld - Home of Project Sun SPOT. Project website avaiable at: http://www.sunspotworld.com/. Last access: 17.3.2009.

[3] The Lex & Yacc Page. Website for compilertools avaiable at: http://dinosaur.compilertools.net/. Last access: 17.3.2009.

[4] Homepage of the TinyOS project, Oktober 2008. Project website available at: http://www.tinyos.net/scoop/. Last access: 17.3.2009.

[5] Hugs Functional Programming System, July 2008. Project website available at: http://www.haskell.org/hugs/. Last access: 17.3.2009.

[6] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In *SIGMOD Conference*, pages 59–68, 1992.

[7] M. Y. Akil. SeNDB: A Sensor Network Database. Master's thesis, Department of Mathematics and Computer Science, Freie Universitaet Berlin, 2006.

[8] J. N. Al-karaki and A. E. Kamal. Routing Techniques in Wireless Sensor Networks: A Survey. *IEEE Wireless Communications*, 11:6–28, 2004.

[9] J. F. Allen. Towards a General Theory of Action and Time. *Artificial Intelligence*, 23(2):123–154, 1984.

[10] J. Ancelin and P. Legaud. An Expert System for Nuclear Reactor Alarm Processing. In *6th Internation Workshop Vol. 1 on Expert Systems & Their Applications*, pages 817–833, Paris La Defense, France, France, 1987. Agence de l'Informatique.

[11] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A Line in the

Sand: A Wireless Sensor Network for Target Detection, Classification, and Tracking. *Computer Networks*, 46(5):605–634, 2004.

[12] A. Arora, M. Gouda, J. Hallstrom, T. Herman, W. Leal, and N. Sridhar. A State-based Language for Sensor-Actuator Networks. *ACM SIGBed Review*, April, 2007.

[13] M. Baar. Homepage of the BMBF project FeuerWhere, July 2008. Project website available at: http://www.feuerwhere.de/. Last access: 17.3.2009.

[14] M. Baar, H. Will, B. Blywis, T. Hillebrandt, A. Liers, G. Wittenburg, and J. Schiller. The ScatterWeb MSB-A2 Platformfor Wireless Sensor Networks. Technical Report TR-B-08-15, Computer Science and Mathematics, Freie Universtät Berlin, 09 2008.

[15] C. Basaran, S. Baydere, G. Bongiovanni, A. Dunkels, O. M. Ergin, L. M. Feeney, I. Hacioglu, V. Handziski, A. KĺOpke, M. Lijding, G. Maselli, N. Meratnia, C. Petrioli, S. Santini, L. van Hoesel, T. Voigt, and A. Zanella. Research Integration: Platform Survey Critical Evaluation of Platforms commonly used in Embedded Wisents Research. Technical report, Embedded WiSeNts consortium, Embedded WiSeNts consortium, June 2006.

[16] R. Beckwith, D. Teibel, and P. Bowen. Report from the Field: Results from an Agricultural Wireless Sensor Network. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 471–478, Washington, DC, USA, 2004. IEEE Computer Society.

[17] M. Beigl, T. Zimmer, A. Krohn, C. Decker, and P. Robinson. Smart-Its – Communication and Sensing Technology for UbiComp Environments. ISSN 1432-7864 2003/2, University of Karlsruhe, 2003.

[18] P. A. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, 1996.

[19] B. Berstel, P. Bonnard, F. Bry, M. Eckert, and P.-L. Patranjan. Reactive Rules on the Web. In G. Antoniou, U. Aßmann, C. Baroglio, S. Decker, N. Henze, P.-L. Patranjan, and R. Tolksdorf, editors, *Reasoning Web*, volume 4636 of *Lecture Notes in Computer Science*, pages 183–239, Dresden, Germany, September 2007. Springer. Third International Summer School 2007, Tutorial Lectures.

[20] J. Beutel. Metrics for Sensor Network Platforms. In *REALWSN '06: Proceedings of the Workshop on Real-world Wireless Sensor Networks*, Uppsala, Sweden, June 2006.

[21] J. Beutel, O. Kasten, and M. Ringwald. BTnodes - A Distributed Platform for Sensor Nodes. In I. F. Akyildiz, D. Estrin, D. E. Culler, and M. B.

Srivastava, editors, *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys03)*, pages 292–293, Los Angeles, California„ USA, November 2003. ACM.

[22] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks*, 10(4):563–579, 2005.

[23] B. Blum, P. Nagaraddi, A. Wood, T. Abdelzaher, S. Son, and J. Stankovic. An Entity Maintenance and Connection Service for Sensor Networks. In *MobiSys '03: Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, pages 201–214, New York, NY, USA, 2003. ACM.

[24] B. Blywis. Entwurf und Implementierung einer modularen und grafisch orientierten Entwirklungsumgebung mit eingebetteter Firmware für den Sensorknoten MSB430. Master's thesis, Department of Mathematics and Computer Science, Freie Universitaet Berlin, 2007.

[25] D. G. Bobrow, S. Mittal, and M. J. Stefik. Expert Systems: Perils and Promise. *Communications of the ACM*, 29(9):880–894, September 1986.

[26] B. G. Buchanan and E. H. Shortliffe. *Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley, Reading, Mass. [u.a.], 1984.

[27] R. Cardell-Oliver, K. Smettem, M. Kranz, and K. Mayer. Field Testing a Wireless Sensor Network for Reactive Environmental. In *Intelligent Sensors, Sensor Networks and Information Processing Conference*, 2004.

[28] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[29] M. Ceriotti, L. Mottola, A. L. Murphy, S. Guna, M. Corr?, M. Pozzi, D. Zonta, and P. Zanon. Monitoring Heritage Buildings with Wireless Sensor Networks: The Torre Aquila Deployment. In *Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2009, SPOTS track)*, San Francisco (CA, USA), April 2009.

[30] S. Chakravarthy. Sentinel: An Object-oriented DBMS with Event-based Rules. *SIGMOD Record*, 26(2):572–575, 1997.

[31] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *VLDB*

'94: Proceedings of the 20th International Conference on Very Large Data Bases, pages 606–617, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[32] S. Chakravarthy, R. Le, and R. Dasari. ECA Rule Processing in Distributed and Heterogeneous Environments. In *DOA '99: Proceedings of the International Symposium on Distributed Objects and Applications*, page 330, Washington, DC, USA, 1999. IEEE Computer Society.

[33] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[34] N. Chomsky. Three Models for the Description of Language. *IRE Transactions on Information Theory*, 2(3):113–124, September 1956.

[35] D. Chu, K. Lin, A. Linares, G. Nguyen, and J. M. Hellerstein. Sdlib: A Sensor Network Data and Communications Library for Rapid and Robust Application Development. In *IPSN '06: Proceedings of the fifth International Conference on Information Processing in Sensor Networks*, pages 432–440, New York, NY, USA, 2006. ACM Press.

[36] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G. P. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis. The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario. In *PERCOM '07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications*, pages 69–78, Washington, DC, USA, 2007. IEEE Computer Society.

[37] P. Costa, G. Coulson, C. Mascolo, G. P. Picco, and S. Zachariadis. The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems. In *Proceedings of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC'05)*, Berlin (Germany), Sept. 2005.

[38] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. Programming Wireless Sensor Networks with the TeenyLIME Middleware. In *Proceedings of the 8th ACM/IFIP/USENIX International Middleware Conference (Middleware 2007)*, Newport Beach (CA, USA), November 2007.

[39] Crossbow. The MicaZ product page. Product website available at: http://www.xbow.com/Products/productdetails.aspx?sid=164. Last access: 17.3.2009.

[40] J. L. David L. Hall, editor. *Handbook of Multisensor Data Fusion*, volume 1. CRC; 1 Edition, June 2001.

[41] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HiPAC Project: Combining Active Databases and Timing Constraints. *SIGMOD Record*, 17(1):51–70, 1988.

[42] C. Decker, A. Krohn, M. Beigl, and T. Zimmer. The Particle Computer System. In *IPSN '05: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, page 62, Piscataway, NJ, USA, 2005. IEEE Press.

[43] F. C. Delicato, P. F. Pires, L. Rust, L. Pirmez, and J. F. de Rezende. Reflective Middleware for Wireless Sensor Networks. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1155–1159, New York, NY, USA, 2005. ACM.

[44] A. Deshpande, C. Guestrin, and S. Madden. Resource-Aware Wireless Sensor-Actuator Networks. *IEEE Data Engineering Bulletin*, 28(1):40–47, 2005.

[45] A. Dunkels. Full TCP/IP for 8 Bit Architectures. In *Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*, San Francisco, May 2003.

[46] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, Nov. 2004.

[47] A. Dunkels, F. Österlind, and Z. He. An adaptive communication architecture for wireless sensor networks. In *Proceedings of the Fifth ACM Conference on Networked Embedded Sensor Systems (SenSys 2007)*, Sydney, Australia, Nov. 2007.

[48] N. Dziengel. Verteilte Ereigniserkennung in Sensornetzen. Master's thesis, Department of Mathematics and Computer Science, Freie Universität Berlin, Oct. 2007.

[49] K. Fall and K. Varadhan. *The ns-2 Manual*. The VINT Project, UC Berkeley, LBL, and Xerox PARC, USC/ISI, December 2007.

[50] Q. Fang, F. Zhao, and L. Guibas. Lightweight Sensing and Communication Protocols for Target Enumeration and Aggregation. In *MobiHoc '03: Proceedings of the 4th ACM International Symposium on Mobile ad hoc Networking & Computing*, pages 165–176, New York, NY, USA, 2003. ACM.

[51] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. In *Proceed-*

ings of the 24th International Conference on Distributed Computing Systems (ICDCS'05), pages 653–662. IEEE, June 2005.

[52] C. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[53] A. Förster and A. Murphy. FROMS: Feedback Routing for Optimizing Multiple Sinks in WSN with Reinforcement Learning. In *Proceedings of the 3rd International Conference on Intelligent Sensors, Sensor Networks, and Information Processing (ISSNIP)*, Melbourne, Australia, December 2007.

[54] A. Förster, A. L. Murphy, J. Schiller, and K. Terfloth. An Efficient Implementation of Reinforcement Learning Based Routing on Real WSN Hardware. In *WIMOB '08: Proceedings of the 2008 IEEE International Conference on Wireless & Mobile Computing, Networking & Communication*, pages 247–252, Washington, DC, USA, 2008. IEEE Computer Society.

[55] C. Frank and K. Römer. Algorithms for Generic Role Assignment in Wireless Sensor Networks. In *Proceedings of the 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys 2005)*, San Diego, CA, USA, Nov. 2005.

[56] E. Friedman-Hill. *Jess in Action : Java Rule-Based Systems (In Action series)*. Manning Publications, December 2002.

[57] F. Galindo and D. Broom. The Relationship between Social Behaviour of Dairy Cows and the Occurrence of Lameness in three Herds. *Research in Veterinary Science*, 69:75–79, 2000.

[58] A. Galton and J. C. Augusto. Two Approaches to Event Definition. In *In Proceedings of the Internatioal Conference on Database and Expert Systems Applications*, pages 547–556. Springer-Verlag, 2002.

[59] T. Gao, C. Pesto, L. Selavo, Y. Chen, J. Ko, J. Lim, A. Terzis, A. Watt, J. Jeng, B. rong Chen, K. Lorincz, and M. Welsh. Wireless Medical Sensor Networks in Emergency Response: Implementation and Pilot Results. In *2008 IEEE International Conference on Technologies for Homeland Security*, May 2008.

[60] S. Gatziu and K. R. Dittrich. SAMOS: an Active Object-Oriented Database System. *IEEE Data Engineering Bulletin*, 15(1-4):23–26, 1992.

[61] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems, 2003.

[62] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event Specification in an Active Object-Oriented Database. *Sigmod Records.*, 21(2):81–90, 1992.

[63] I. Genuth. The Future of Electronic Paper. http://thefutureofthings.com/articles/1000/the-future-of-electronic-paper.html, October 2007. Last access: 17.3.2009.

[64] M. Goralczyk. Development and Evaluation of Hop-Based Topology Algorithms For Wireless Sensor Networks. Master's thesis, Department of Mathematics and Computer Science, Freie Universitaet Berlin, 2006.

[65] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification*. The Java Series. Prentice Hall, 2nd edition, 2000.

[66] P. Graham. *Hackers and Painters: Big Ideas from the Computer Age*. O'Reilly, May 2004.

[67] B. Greenstein, E. Kohler, and D. Estrin. A Sensor Network Application Construction Kit (SNACK). In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 69–80, New York, NY, USA, 2004. ACM.

[68] L. Gygax, G. Neisen, and H. Bollhalder. Accuracy and Validation of a Radar-based Automatic Local Position Measurement System for Tracking Dairy Cows in free-stall Barns. *Computational Electronic Agriculture*, 56(1):23–33, 2007.

[69] G. Hackmann, C.-L. Fok, G.-C. Roman, and C. Lu. Agimone: Middleware Support for Seamless Integration of Sensor and IP Networks. In *Lecture Notes in Computer Science*, volume 4026, pages 101–118, 2006.

[70] T. Haenselmann, T. King, M. Busse, W. Effelsberg, and M. Fuchs. Scriptable Sensor Network Based Home-Automation. In *EUC Workshops*, pages 579–591, 2007.

[71] A. Hagedorn, D. Starobinski, and A. Trachtenberg. Rateless Deluge: Over-the-Air Programming of Wireless Sensor Networks Using Random Linear Codes. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks*, pages 457–466, Washington, DC, USA, 2008. IEEE Computer Society.

[72] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. In *MobiSys '05: Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, pages 163–176, New York, NY, USA, 2005. ACM Press.

[73] T. He, S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J. A. Stankovic, T. F. Abdelzaher, J. Hui, and B. H. Krogh. VigilNet: An Integrated Sensor Network System for Energy-efficient Surveillance. *TOSN*, 2(1):1–38, 2006.

[74] W. Heinzelman, A. Murphy, H. Carvalho, and M. Perillo. Middleware to Support Sensor Network Applications. *IEEE Network*, 18:6–14, 2004.

[75] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In *Proccedings of the Hawaii Conference on System Sciences (HICSS)*, Washington, DC, USA, 2000. IEEE Computer Society.

[76] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language*. Microsoft .NET Development Series. Addison-Wesley, 2003.

[77] J. Hill. Spec mote website. Project website available at: http://www.jlhlabs.com/jhill_cs/spec/. Last access: 17.3.2009.

[78] J. Hill and D. Culler. A Wireless Embedded Sensor Architecture for System-level Optimization. Technical report, U.C. Berkeley, 2001.

[79] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[80] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-calculus*. Cambridge University Press, New York, NY, USA, 1986.

[81] L. Hu and D. Evans. Localization for Mobile Sensor Networks. In *MobiCom '04: Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, pages 45–57, New York, NY, USA, 2004. ACM Press.

[82] J. W. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM.

[83] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Mobile Computing and Networking*, pages 56–67, 2000.

[84] A. Jäger. Entwicklung eines Schnittstellen- und Modulkonzeptes für das regelbasierte Middleware Framework FACTS. Master's thesis, Department of Mathematics and Computer Science, Freie Universitaet Berlin, 2007.

[85] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 37, pages 96–107, New York, NY, USA, October 2002. ACM Press.

[86] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next Century Challenges: Mobile Networking for Smart Dust. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 271–278, New York, NY, USA, 1999. ACM.

[87] O. Kasten. *A State-Based Programming Model for Wireless Sensor Networks*. PhD thesis, ETH Zurich, Zurich, Switzerland, June 2007.

[88] O. Kasten and K. Römer. Beyond Event Handlers: Programming Wireless Sensors with Attributed State Machines. In *The Fourth International Conference on Information Processing in Sensor Networks (IPSN)*, pages 45–52, Los Angeles, USA, Apr. 2005.

[89] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health Monitoring of Civil Infrastructures using Wireless Sensor Networks. In *IPSN '07: Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, pages 254–263, New York, NY, USA, 2007. ACM.

[90] J. Koshy and R. Pandey. VMSTAR: Synthesizing Scalable Runtime Environments for Sensor Networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 243–254, New York, NY, USA, 2005. ACM.

[91] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and Efficient Programming Abstractions for Wireless Sensor Networks. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 200–210, New York, NY, USA, 2007. ACM Press.

[92] R. A. Kowalski and M. J. Sergot. A Logic-Based Calculus of Events. In *Foundations of Knowledge Base Management (Xania)*, pages 23–55, 1985.

[93] P. Levis and D. Culler. Mate: A Tiny Virtual Machine for Sensor Networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems.*, San Jose, CA, USA, October 2002.

[94] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: a Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[95] P. Lewis, D. Gay, and D. Culler. Bridging the Gap: Programming Sensor Networks with Application Specific Virtual Machines. Techical Report CSD-04-1343, University of California at Berkeley, 2004.

[96] A. Liers. Homepage of the ScatterWeb project, Oktober 2008. Project Website available at: http://cst.mi.fu-berlin.de/projects/ScatterWeb/. Last access: 17.3.2009.

[97] T. Liu and M. Martonosi. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In *In PPoPP ?03: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[98] T. Liu, C. M. Sadler, P. Zhang, and M. Martonosi. Implementing Software on Resource-constrained Mobile Sensors: Experiences with Impala and ZebraNet. In *MobiSys '04: Proceedings of the 2nd International Conference on Mobile systems, Applications, and Services*, pages 256–269, New York, NY, USA, 2004. ACM.

[99] S. R. Madden, J. M. Hellerstein, and W. Hong. TinyDB: In-network Query Processing in TinyOS. Release Notes, September 2003. http://telegraph.cs.berkeley.edu/tinydb/tinydb.pdf.

[100] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, September 2002.

[101] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks. In *Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN 2006)*, pages 212–227, February 2006.

[102] P. J. Marrón, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, and K. Rothermel. TinyCubus: A Flexible and Adaptive Framework for Sensor Networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, pages 278–289, January 2005.

[103] K. Martinez, P. Padhy, A. Riddoch, R. Ong, and J. Hart. Glacial Environment Monitoring using Sensor Networks. In *Proceedings of the Workshop on Real-World Wireless Sensor Networks (REALWSN'05)*, Stockholm, Sweden, June 2005.

[104] J. P. McDermott. R1: A Rule-Based Configurer of Computer Systems. *Artificial Intelligence*, 19(1):39–88, 1982.

[105] L. Mottola and G. P. Picco. Logical Neighborhoods: A Programming Abstraction for Wireless Sensor Networks. In P. Gibbons, T. Abdelzaher, J. Aspnes, and R. Rao, editors, *Proceedings of the 2nd International Conference on Distributed Computing in Sensor Systems (DCOSS)*, number 4026 in Lecture Notes on Computer Science, pages 150–167, San Francisco (CA, USA), June 2006. Springer.

[106] T. Naumowicz, R. Freeman, A. Heil, M. Calsyn, E. Hellmich, A. Brändle, T. Guilford, and J. Schiller. Autonomous Monitoring of Vulnerable Habitats using a Wireless Sensor Network. In *REALWSN '08: Proceedings of the workshop on Real-world wireless sensor networks*, pages 51–55, New York, NY, USA, 2008. ACM.

[107] R. Newton, Arvind, and M. Welsh. Building up to Macroprogramming: An Intermediate Language for Sensor Networks. In *IPSN '05: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, Piscataway, NJ, USA, 2005. IEEE Press.

[108] R. Newton, G. Morrisett, and M. Welsh. The Regiment Macroprogramming System. In T. F. Abdelzaher, L. J. Guibas, and M. Welsh, editors, *IPSN: Proceedings of the 6th International Symposium on Information Processing in Sensor Networks*, pages 489–498. ACM, 2007.

[109] R. Newton and M. Welsh. Region Streams: Functional Macroprogramming for Sensor Networks. In *DMSN '04: Proceeedings of the 1st International Workshop on Data Management for Sensor Networks*, pages 78–87, New York, NY, USA, 2004. ACM Press.

[110] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[111] A. Paschke. ECA-RuleML: An Approach combining ECA Rules with temporal interval-based KR Event/Action Logics and Transactional Update Logics. Proposal for RuleML Reaction Rules Technical Group, 2005.

[112] A. Pathak, L. Mottola, A. Bakshi, V. K. Prasanna, and G. P. Picco. Expressing Sensor Network Interaction Patterns Using Data-Driven Macroprogramming. In *PERCOMW '07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 255–260, Washington, DC, USA, 2007. IEEE Computer Society.

[113] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler. SPINS: Security Protocols for Sensor Networks. *Wireless Networks*, 8(5):521–534, September 2002.

[114] K. Pister. Smart Dust - Autonomous sensing and communication in a cubic millimeter. Website avaiable at: http://robotics.eecs.berkeley.edu/ pister/SmartDust/. Last access: 17.3.2009.

[115] O. G. Ramakrishna Gummadi and R. Govindan. Macro-programming Wireless Sensor Networks using Kairos. In *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2005.

[116] D. R. Raymond and S. F. Midkiff. Denial-of-Service in Wireless Sensor Networks: Attacks and Defenses. *Pervasive Computing, IEEE*, 7(1):74–81, 2008.

[117] O. Richter. Entwurf und Implementierung einer virtuellen Maschine für Sensornetze mit Unterstützung für imperative und regelbasierte Programmierung. Master's thesis, Department of Mathematics and Computer Science, Freie Universität Berlin, 2006.

[118] R. Riem-Vis. Cold Chain Management using an Ultra Low Power Wireless Sensor Network. In *In WAMES*, Boston, USA, 2004.

[119] D. Rogoz and F. O?Reilly. *Ambient Intelligence with Microsystems*, volume 18, chapter Dedicated Networking Solutions for a Container Tracking System, pages 387–408. Springer US, 2009.

[120] K. Römer. Programmierung von Sensornetzen. Presentation at KuVS Summerschool "Sensornetze - Brücke zwischen Virtualität und Realität", September 2004. Available at: http://www.ibr.cs.tu-bs.de/news/ibr/summerschool2004/roemer-sysprog.pdf. Last access: 17.3.2009.

[121] K. Römer. Programming Paradigms and Middleware for Sensor Networks. *GI/ITG Fachgespräch Sensornetze, Karlsruhe*, Feb. 2004.

[122] K. Römer. Discovery of Frequent Distributed Event Patterns in Sensor Networks. In *European Conference on Wireless Sensor Networks (EWSN 2008)*, pages 106–124, Bologna, Italy, Jan. 2008.

[123] K. Römer, C. Frank, P. J. Marrón, and C. Becker. Generic Role Assignment for Wireless Sensor Networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, pages 7–12, Leuven, Belgium, September 2004.

[124] K. Römer, O. Kasten, and F. Mattern. Middleware Challenges for Wireless Sensor Networks. *SIGMOBILE Mobile Computer Communications Review*, 6(4):59–61, 2002.

[125] S. Roundy, D. Steingart, L. Frechette, P. K. Wright, and J. M. Rabaey. Power Sources for Wireless Sensor Networks. In H. Karl, A. Willig, and A. Wolisz, editors, *Wireless Sensor Networks, First European Workshop (EWSN 2004)*, volume 2920, pages 1–17, Berlin, Germany, January 2004. Springer.

[126] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, 1986.

[127] J. Schulz, F. Reichenbach, J. Blumenthal, and D. Timmermann. Low Cost System for Detecting Leakages along Artificial Dikes with Wireless Sensor

Networks. In *REALWSN '08: Proceedings of the workshop on Real-world wireless sensor networks*, pages 66–70, New York, NY, USA, 2008. ACM.

[128] S. Sen and R. Cardell-Oliver. A Rule-Based Language for Programming Wireless Sensor Actuator Networks using Frequency and Communication. In *Proceedings of the 3th Workshop on Embedded Networked Sensors (EmNets)*, Cambridge, USA, May 2006.

[129] P. Sikka, P. Corke, P. Valencia, C. Crossman, D. Swain, and G. Bishop-Hurley. Wireless Adhoc Sensor and Actuator Networks on the Farm. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks*, pages 492–499, New York, NY, USA, 2006. ACM.

[130] D. Simon and C. Cifuentes. The Squawk Virtual Machine: Java on the bare metal. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 150–151, New York, NY, USA, 2005. ACM.

[131] P. Stanley-Marbell and L. Iftode. Scylla: A Smart Virtual Machine for Mobile Embedded Systems. In *3rd IEEE Workshop on Mobile Computing Systems and Applications, WMCSA2000*, December 2000.

[132] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1981.

[133] R. Sugihara and R. K. Gupta. Programming Models for Sensor Networks: A Survey. *ACM Transactions on Sensor Networks*, 4(2):1–29, March 2008.

[134] I. Talzi, A. Hasler, S. Gruber, and C. Tschudin. PermaSense: Investigating Permafrost with a WSN in the Swiss Alps. In *EmNets '07: Proceedings of the 4th Workshop on Embedded Networked Sensors*, pages 8–12, New York, NY, USA, 2007. ACM.

[135] D. Tennenhouse. Proactive Computing. *Communications of the ACM*, 43(5):43–50, 2000.

[136] K. Terfloth. Homepage of the FACTS Middleware Framework, January 2009. Project website available at: http://cst.mi.fu-berlin.de/projects/FACTS/index.html. Last access: 17.3.2009.

[137] J. H. Thanos Stathopoulos and D. Estrin. A Remote Code Update Mechanism for Wireless Sensor Networks. Technical Report 30, CENS UCLA, 2003.

[138] S. Thompson. *Haskell: The Craft of Functional Programming (2nd Edition)*. Addison Wesley, March 1999.

[139] S. Tilak, A. Murphy, and W. Heinzelman. Non-Uniform Information Dissemination for Sensor Networks. In *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*, page 295, Washington, DC, USA, 2003. IEEE Computer Society.

[140] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A Macroscope in the Redwoods. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 51–63, New York, NY, USA, 2005. ACM Press.

[141] N. Tsiftes, A. Dunkels, and T. Voigt. Efficient Sensor Network Reprogramming through Compression of Executable Modules. In *Proceedings of the Fifth Annual IEEE Communications Society Conference on Sensor, Mesh, and Ad Hoc Communications and Networks*, June 2008.

[142] F. L. Villafuerte, K. Terfloth, and J. H. Schiller. Using Network Density as a New Parameter to Estimate Distance. In *Proceedings of the Seventh International Conference on Networking (ICN 2008)*, pages 30–35, Cancun, Mexico, April 2008.

[143] T. Voigt, H. Ritter, and J. Schiller. Utilizing Solar Power in Wireless Sensor Networks. In *LCN '03: Proceedings of the 28th Annual IEEE International Conference on Local Computer Networks*, page 416, Washington, DC, USA, 2003. IEEE Computer Society.

[144] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Nessages: A Mechanism for Integrated Communication and Computation. In *In Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, 1992.

[145] B. Warneke, M. Last, B. Liebowitz, and K. S. J. Pister. Smart Dust: Communicating with a Cubic-Millimeter Computer. *Computer*, 34(1):44–51, 2001.

[146] C. Wartenburger. Experimentelle Analyse verteilter Ereigniserkennung in Sensornetzen. Master's thesis, Department of Mathematics and Computer Science, Freie Universität Berlin, Dec. 2008.

[147] S. T. Wei Yuan, S.V. Krishnamurthy. Synchronization of Multiple Levels of Data Fusion in Wireless Sensor Networks. In *Global Telecommunications Conference. IEEE GLOBECOM '03.*, 2003.

[148] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *1st Symposium on Networked Systems Design and Implementation (NSDI 2004)*, pages 29–42, March 2004.

[149] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and Yield in a Volcano Monitoring Sensor Network. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 27–27, Berkeley, CA, USA, 2006. USENIX Association.

[150] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A Neighborhood Abstraction for Sensor Networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 99–110, New York, NY, USA, 2004. ACM Press.

[151] E. C. Whitman. SOSUS; The "Secret Weapon" of Undersea Surveillance. *Undersea Warfare*, 7(2), 2005.

[152] J. Widom. The Starburst Active Database Rule System. *IEEE Transactions on Knowledge and Data Engineering*, 8:583–595, 1996.

[153] G. Wittenburg. A Rule-Based Middleware Architecture for Wireless Sensor Networks. Master's thesis, Department of Mathematics and Computer Science, Freie Universitaet Berlin, Nov. 2005.

[154] G. Wittenburg, K. Terfloth, F. L. Villafuerte, T. Naumowicz, H. Ritter, and J. Schiller. Fence Monitoring - Experimental Evaluation of a Use Case for Wireless Sensor Networks. In *Proceedings of the Fourth European Conference on Wireless Sensor Networks (EWSN '07)*, pages 163–178, Delft, The Netherlands, Jan. 2007.

[155] A. D. Wood and J. A. Stankovic. Denial of Service in Sensor Networks. *Computer*, 35(10):54–62, 2002.

[156] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A Wireless Sensor Network for Structural Monitoring. In *In Proceedings of the ACM Conference on Embedded Networked Sensor Systems(Sensys04)*, November 2004.

[157] Y. Xu, J. Heidemann, and D. Estrin. Geography-informed Energy Conservation for Ad Hoc Routing. In *MobiCom '01: Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 70–84, New York, NY, USA, 2001. ACM.

[158] Y. Yao and J. E. Gehrke. The Cougar Approach to in-network Query Processing in Sensor Networks. *ACM SIGMOD Record*, 31(2):9–18, Sept. 2002.

Appendices

Appendix A

The RDL language grammar

ruleset	::=	“**ruleset**” identifier block_list
identifier	::=	[a−zA−Z_][a−zA−Z_\−0−9]∗
block_list	::=	block { block }
block	::=	named_name
	|	named_slot
	|	fact
	|	rule
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
                        |  “send” expression expression slot
                        |  “set” slot “=” expression
                        |  “flush” slot
                        |  “touch” 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”
                        |  “~”
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

Source code of selected rulesets

This appendix features a selection of rulesets implemented for evaluation by FACTS and tested on the ScatterWeb MSB430 sensor nodes. Sources are freely available for download on the project website [136].

B.1 The Xmas ruleset

Listing B.1: The Xmas program, coordinated enlightment.

```
1
2  ruleset ChristmasLights
3
4  /*
5   * Building an MSP430 christmas tree
6   * For decent christmas decoration
7   * (1) Insert new batteries into the nodes
8   * (2) Follow the staggering stacking instructions below and
9   * (3) Apply this ruleset
10  *
11  *          |1|                    (master node, id = 1)
12  *        |2| |2|                  (nodes in 2nd row, id = 2)
13  *      |3| |3| |3|                (nodes in 3rd row,  id = 3)
14  *    |4| |4| |4| |4|
15  *          ...                    (supply all nodes available)
16  *
17  * Trigger enlightment with sending a bootstrap fact to the
        master
18  *
19  */
20
21 name bootstrap = "bootstrap"
22 name timer_context = "timer_context"
23 name random = "random"
24
25 name system = "system"
26 name row = "row"
27 name row_reply = "row_reply"
```

```
28 name trigger = "trigger"
29 name delay = "delay"
30 name counter = "counter"
31 name light = "light"
32 name ON = "ON"
33 name node = "node"
34
35 fact system [broadcast = 0, tx-range = 10]
36
37 slot systemBroadcast = {system broadcast}
38 slot systemTxRange = {system tx-range}
39 slot systemID = {system owner}
40
41 //all the next rules coordinate master behavior
42
43 rule getNumRows 100
44 <- exists {bootstrap}
45 -> define row [depth = 0]
46 -> define light [ON = true, node = 0]
47 -> send systemBroadcast systemTxRange {row}
48
49 rule getMaxNum 100
50 <- exists {row_reply
51     <- eval ({this depth} > {row depth})}
52 -> set {row depth} = {row_reply depth}
53 -> retract {row_reply}
54 -> call removeTimer ({delay})
55 -> call setTimer ({delay}, 1)
56
57
58 rule replyInitiate 99
59 <- exists {delay}
60 <- eval ({row depth} > 0)
61 <- eval (systemID == 1)
62 -> define timer_context [interval = 1, ident = "trigger"]
63 -> define counter [current = 0]
64 -> retract {delay}
65
66 rule selfEnlight 98
67 <- exists {trigger}
68 <- eval ({counter current} == 0)
69 -> set {counter current} = ({counter current} + 1)
70 -> call setLED (255)
71 -> set {light ON} = true
72 -> retract {trigger}
73
74 rule sendON 98
75 <- exists {trigger}
76 <- eval ({counter current} != 0)
77 <- eval ({counter current} < {row depth})
78 -> set {counter current} = ({counter current} + 1)
79 -> set {light node} = {counter current}
80 -> send systemBroadcast systemTxRange {light}
81 -> retract {trigger}
```

```
82
83  rule sendOFF 97
84  <- exists {trigger}
85  <- eval ({counter current} >= {row depth})
86  -> set {counter current} = 0
87  -> set {light ON} = false
88  -> call setLED (250)
89  -> send systemBroadcast systemTxRange {light}
90  -> set {light node} = 0
91  -> retract {trigger}
92
93  //the following rules take care of slave behavior
94  rule replyDepth 80
95  <- exists {row}
96  <- eval (systemID != 1)
97  -> define row_reply [depth = systemID]
98  -> send 1 systemTxRange {row_reply}
99  -> retract {row_reply}
100 -> retract {row}
101
102 rule setON 79
103 <- exists {light
104     <- eval ({this ON} == true)
105     <- eval ({this node} == systemID)}
106 <- eval (systemID != 1)
107 -> call setLED (255)
108 -> retract {light}
109
110 rule setOFF 78
111 <- exists {light
112     <- eval ({this ON} == false)}
113 <- eval (systemID != 1)
114 -> call setLED (250)
115 -> retract {light}
```

B.2 The Directed Diffusion ruleset

Listing B.2: Directed Diffusion, one-phase pull variant

```
1
2  ruleset DirectedDiffusion
3
4  /*
5   * This implementation features two sinks (nodes 1 and 2)
6   * and one data source (node 6)
7   * Sink announcements, thus interests are issued every 15 seconds
8   * in case no data has yet arrived afterwards,
9   * the route maintenance interval is set to 50 seconds
10  * In this time, 100 packets are issued by the source
11  * targeting both sinks
12  */
13
14 name data = "data"
15 name system = "system"
16 name intervals = "intervals"
17 name stats = "stats"
18 name bootstrap = "bootstrap"
19
20 /*
21  * The following two facts serve as a constants
22  * for system related parameters, as well as to
23  * parametrize the timing and packet concerns
24  */
25
26 fact intervals [annum = 0, pre_sink = 15, data = 5, post_sink =
       50]
27 fact system [broadcast = 0, tx-range = 10, counter = 0, cur_id =
       0]
28
29 slot systemID = {system owner}
30 slot systemBroadcast = {system broadcast}
31 slot systemTxRange = {system tx-range}
32
33 slot ddAnn = {intervals annum}
34 slot pktCurID = {system cur_id}
35
36 slot preSink = {intervals pre_sink}
37 slot dataInt = {intervals data}
38 slot postSink = {intervals post_sink}
39
40 //Declaration of directed diffusion specific names
41
42 name interest = "interest"
43 name gradient = "gradient"
44
45 name retransmitSATimerExpired = "retransmitSATimerExpired"
46 name dataTimer = "dataTimer"
47 name triggerResendTimer = "triggerResendTimer"
48
```

```
49
50 /*
51  * Dissemination of Interests by the sink nodes
52  * The interval at which these announcements are spread changes
53  * upon reception of the first data packet from a source node
54  * These rules are solely triggered at the sinks
55  */
56
57
58 rule disseminateInterests 100
59 <- exists {bootstrap}
60 <- eval (systemID < 3)
61 -> define interest [ann = ddAnn, sink = systemID, weight = 1,
       neighbor = systemID]
62 -> send systemBroadcast systemTxRange {interest}
63 -> retract {interest}
64 -> retract {bootstrap}
65 -> call setTimer({retransmitSATimerExpired}, preSink)
66
67
68
69 rule retransmitInterests 99
70 <- exists {retransmitSATimerExpired}
71 <- eval ({system counter} == 0)
72 -> retract {retransmitSATimerExpired}
73 -> set ddAnn = (ddAnn + 1)
74 -> define interest [ann = ddAnn, sink = systemID, weight = 1,
       neighbor = systemID]
75 -> send systemBroadcast systemTxRange {interest}
76 -> retract {interest}
77 -> call setTimer({retransmitSATimerExpired}, preSink)
78
79
80
81 rule periodSinkHelloInit 98
82 <- exists {data}
83 <- eval (systemID < 3)
84 <- eval ({system counter} == 0)
85 -> set ddAnn = (ddAnn + 1)
86 -> call setTimer({retransmitSATimerExpired}, postSink)
87
88
89
90 rule periodicSinkHello 97
91 <- exists {retransmitSATimerExpired}
92 -> retract {retransmitSATimerExpired}
93 -> define interest [ann = ddAnn, sink = systemID, weight = 1,
       neighbor = systemID]
94 -> send systemBroadcast systemTxRange {interest}
95 -> retract {interest}
96 -> set ddAnn = (ddAnn + 1)
97 -> call setTimer({retransmitSATimerExpired}, postSink)
98
99
```

```
100 /*
101  * Interest handling rules.
102  * These are applied by all nodes in the network
103  */
104
105
106 rule dropInterestsWhenSink 96
107 <- exists {interest
108   <- eval ({this sink} == systemID)}
109 -> retract {interest}
110
111
112 rule reinforceInterests 95
113 <- exists {interest}
114 <- exists {gradient
115   <- eval ({this sink} == {interest sink})
116   <- eval ({this weight} > {interest weight})
117 }
118 -> set {gradient weight
119   <- eval ({this sink} == {interest sink})} = {interest weight}
120 -> set {gradient neighbor
121   <- eval ({this sink} == {interest sink})} = {interest neighbor}
122 -> set {gradient last_ann
123   <- eval ({this sink} == {interest sink})} = {interest ann}
124 -> set {interest weight} = ({interest weight} + 1)
125 -> set {interest neighbor} = systemID
126 -> send systemBroadcast systemTxRange {interest}
127 -> retract {interest}
128
129
130 rule removeDuplicateInterests 94
131 <- exists {interest}
132 <- exists {gradient
133   <- eval ({this sink} == {interest sink})
134         <- eval ({this last_ann} >= {interest ann})
135 }
136 -> retract {interest}
137
138
139 rule resendInterestForRobustness 93
140 <- exists {interest}
141 <- exists {gradient
142   <- eval ({this sink} == {interest sink})
143         <- eval ({this last_ann} < {interest ann})
144 }
145 -> set {gradient last_ann
146         <- eval ({this sink} == {interest sink})} = {interest ann
              }
147 -> set {interest weight} = ({interest weight} + 1)
148 -> set {interest neighbor} = systemID
149 -> send systemBroadcast systemTxRange {interest}
150 -> retract {interest}
151
152
```

```
153 /*
154  * When a new interested has been received, create a new gradient
155  * for it and broadcast it to the network.
156  */
157
158 rule handleInterests 92
159 <- exists {interest}
160 -> define gradient [sink = {interest sink}, neighbor = {interest
        neighbor}, weight = {interest weight}, last_ann = {interest
        ann}]
161 -> set {interest neighbor} = systemID
162 -> set {interest weight} = ({interest weight} + 1)
163 -> send systemBroadcast systemTxRange {interest}
164 -> retract {interest}
165
166
167 //Slots identifying gradients to sink_1 and sink_2
168
169 slot route_sink_1 = {gradient neighbor
170       <- eval ({this sink} == 1)}
171
172 slot route_sink_2 = {gradient neighbor
173       <- eval ({this sink} == 2)}
174
175
176 /*
177  * Rules for data creation at the source node
178  */
179
180 rule produceData 91
181 <- exists {bootstrap}
182 <- eval (systemID == 6)
183 -> set {system counter} = 100
184 -> retract {bootstrap}
185 -> define data [pkt_id = ddAnn, sink1NB = route_sink_1, sink2NB =
        route_sink_2, numHops = 1]
186 -> send systemBroadcast systemTxRange {data}
187 -> retract {data}
188 -> set ddAnn = (ddAnn + 1)
189 -> call setTimer({dataTimer}, dataInt)
190
191
192 rule sendData 90
193 <- exists {dataTimer}
194 <- eval (ddAnn <= {system counter})
195 -> retract {dataTimer}
196 -> define data [pkt_id = ddAnn, sink1NB = route_sink_1, sink2NB =
        route_sink_2, numHops = 1]
197 -> send systemBroadcast systemTxRange {data}
198 -> retract {data}
199 -> set ddAnn = (ddAnn + 1)
200 -> call setTimer({dataTimer}, dataInt)
201
202
```

```
203
204
205 /*
206  * Data handling rules
207  *
208  * If a node receives a data paket, it will transmit it broadcast
209  * to the two sinks in question. Therefore, it will consults its
210  * gradients and set the routing info accordingly
211  */
212
213 rule handleDataAtSource 90
214 <- exists {data}
215 <- eval(systemID == 6)
216 -> retract {data}
217
218
219 rule handleNewDataAtSink 90
220 <- exists {data}
221 <- eval (systemID <= 2)
222 <- eval ({data pkt_id} > pktCurID)
223 -> set {system counter} = ({system counter} + 1)
224 -> set pktCurID = {data pkt_id}
225
226
227 rule dropDataNotForMe 89
228 <- exists {data}
229 <- eval ({data sink1NB} != systemID)
230 <- eval ({data sink2NB} != systemID)
231 -> retract {data}
232
233
234 rule setRouteToSink12 87
235 <- exists {data}
236 <- eval ({data sink1NB} == systemID)
237 <- eval ({data sink2NB} == systemID)
238 <- eval (systemID != 1)
239 <- eval (systemID != 2)
240 -> set {data sink1NB} = route_sink_1
241 -> set {data sink2NB} = route_sink_2
242 -> set {data numHops} = ({data numHops} + 1)
243 -> send systemBroadcast systemTxRange {data}
244 -> retract {data}
245
246
247 rule setRouteToSink1 87
248 <- exists {data}
249 <- eval ({data sink1NB} == systemID)
250 <- eval (systemID != 1)
251 -> set {data sink1NB} = route_sink_1
252 -> set {data numHops} = ({data numHops} + 1)
253 -> send systemBroadcast systemTxRange {data}
254 -> retract {data}
255
256
```

```
257 rule routeDataSink2 87
258 <- exists {data}
259 <- eval ({data sink2NB} == systemID)
260 <- eval (systemID != 2)
261 -> set {data sink2NB} = route_sink_2
262 -> set {data numHops} = ({data numHops} + 1)
263 -> send systemBroadcast systemTxRange {data}
264 -> retract {data}
265
266
267 rule retractData 86
268 <- exists {data}
269 -> retract {data}
```

B.3 The FROMS routing protocol

Listing B.3: The FROMS routing ruleset

```
1  ruleset FROMS
2
3  name system = "system"
4  name bootstrap = "bootstrap"
5  name PST_node = "PST_node"
6  name cachedMinCostSink = "cachedMinCostSink"
7  name basic_sink = "basic_sink"
8  name timer_context = "timer_context"
9  name data = "data"
10 name unmergedYet = "unmergedYet"
11 name tmp = "tmp"
12
13 fact system [broadcast = 0, tx-range = 20, pst-nodeId = 1,
       counter = 0, pkt_id = 0]
14 fact unmergedYet
15
16 slot systemID = {system owner}
17 slot systemBroadcast = {system broadcast}
18 slot systemTxRange = {system tx-range}
19 slot pstNodeId = {system pst-nodeId}
20
21
22 /*
23  * Phase 1: Sink announcements and PST_nodes setup
24  * the sink nodes (1 and 2) start dissemination of announcements
25  * every 15 seconds as long as no data arrives, they will repeat
26  * their requests  which is controlled via the unmergedYet fact
27  * retracted after the first reception of data
28  * these rules are only executed by the sinks
29  */
30
31 //------ sink_announce [sink, numHops, ttl] ------
32
33 name sink_announce = "sink_announce"
34 name retransmitSATimerExpired = "retransmitSATimerExpired"
35 name sendData = "sendData"
36 name stats = "stats"
37
38 slot sinkSink = {sink_announce sink}
39 slot sinkNumHops = {sink_announce numHops}
40 slot sinkSender = {sink_announce owner}
41 slot sinkTTL = {sink_announce ttl}
42
43
44 rule bootstrap 250
45 <- exists {bootstrap}
46 <- eval (systemID < 3)
47 -> define timer_context [interval = 15, ident = "
       retransmitSATimerExpired"]
48 -> define stats [current = 0, received = 0]
```

```
49
50 rule retransmitSinkAnnouncements 249
51 <- exists {retransmitSATimerExpired}
52 <- exists {unmergedYet}
53 -> retract {retransmitSATimerExpired}
54 -> define sink_announce [sink = systemID, numHops = 1, ttl = 10]
55 -> send systemBroadcast systemTxRange {sink_announce}
56 -> retract {sink_announce}
57
58 rule removeSinkAnnouncementTrigger 248
59 <- exists {retransmitSATimerExpired}
60 -> retract {retransmitSATimerExpired}
61 -> retract {timer_context}
62
63
64 /*
65  * the source (assigned to node 6) periodically starts
66  * disseminating data packets every 5 seconds
67  * these rules are only applied  by the source node
68  */
69
70 //------ data [pkt_id, sink1, sink2, feedback, from] ------
71
72
73 name dataTimer = "dataTimer"
74 name paket = "paket"
75
76
77 rule produceDataTimer 247
78 <- exists {bootstrap}
79 <- eval (systemID == 6)
80 -> set {system counter} = 1
81 -> retract {bootstrap}
82 -> define paket [counter = 1]
83 -> call setTimer({dataTimer}, 5)
84
85
86 rule sendData 246
87 <- exists {dataTimer}
88 <- eval ({paket counter} <= 100)
89 -> retract {dataTimer}
90 -> define data [pkt_id = {paket counter}, sink1 = 6, sink2 = 6,
       feedback = 1, from = 6]
91 -> set {paket counter} = ({paket counter} + 1)
92 -> call setTimer({dataTimer}, 5)
93
94
95 /*
96  * whenever a sink_announcement is coming in that features a new
97  * sink not seen before, generate a PST_node and
98  * retransmit the sink_announcement, otherwise
99  * see whether information stays within bounds, keep or discard
100 */
101
```

```
102 //--- PST_node [unique_id, sink, neighbor_id, cost, flag] ---
103
104
105 slot cachedMinCostSinkFilter = {cachedMinCostSink cost
106   <- eval ({this sink} == sinkSink)}
107
108 slot PST_SS_SN_cost = {PST_node cost
109   <- eval ({this sink} == sinkSink)
110   <- eval ({this neighbor_id} == sinkSender)}
111
112
113 rule dropAnnouncementWhenOriginator 240
114 <- exists {sink_announce
115   <- eval ({this sink} == systemID)}
116 -> retract {sink_announce}
117
118
119 rule purgeOldSinkAnnouncements 239
120 <- exists {sink_announce
121   <- eval ({this ttl} == 0)}
122 -> retract {sink_announce}
123
124
125 rule generatePSTNodeNew 238
126 <- exists {sink_announce
127   <- eval ({this sink} != {PST_node sink})}
128 -> define PST_node [unique_id = pstNodeId, sink = sinkSink,
      neighbor_id = sinkSender, cost = sinkNumHops, flag = 0]
129 -> define cachedMinCostSink [sink = sinkSink, cost = sinkNumHops]
130 -> set pstNodeId = (pstNodeId + 1)
131 -> set sinkNumHops = (sinkNumHops + 1)
132 -> set sinkTTL = (sinkTTL - 1)
133 -> send systemBroadcast systemTxRange {sink_announce}
134 -> retract {sink_announce}
135
136
137 rule updateCachedMinCostRetransmit 237
138 <- exists {sink_announce
139   <- eval ({this numHops} <= cachedMinCostSinkFilter)}
140 -> set cachedMinCostSinkFilter = sinkNumHops
141 -> define sink_announce [sink = sinkSink, numHops = (sinkNumHops
      + 1), ttl = (sinkTTL - 1)]
142 -> send systemBroadcast systemTxRange {sink_announce
143         <- eval ({this owner} == systemID)}
144 -> retract {sink_announce
145         <- eval ({this owner} == systemID)}
146
147
148 rule updatePSTNode 236
149 <- exists {sink_announce}
150 <- eval (PST_SS_SN_cost > sinkNumHops)
151 -> set PST_SS_SN_cost = sinkNumHops
152 -> retract {sink_announce}
153
```

```
154 rule purgeDuplicate 235
155 <- exists {sink_announce}
156 <- eval (PST_SS_SN_cost <= sinkNumHops)
157 -> retract {sink_announce}
158
159 rule sameSinkDifferentNeighbor 234
160 <- exists {sink_announce
161   <- eval ({this numHops} <= (cachedMinCostSinkFilter + 1))}
162 -> define PST_node [unique_id = pstNodeId, sink = sinkSink,
        neighbor_id = sinkSender, cost = sinkNumHops, flag = 0]
163 -> set pstNodeId = (pstNodeId + 1)
164
165
166 rule purgeSinkAnnouncements 233
167 <- exists {sink_announce}
168 -> retract {sink_announce}
169
170
171 /*
172  * Phase 2: Setup Routing Tables  = PSTnodes (path sharing nodes)
173  * This is done by first generating new PSTnodes for shared
174  * routes, with the arrival of first data packet
175  * merged PSTs have a path to different sinks using the same
176  * neighbor; sinks are therefore bitwise "OR"ed to denote a
177  * sharing instance; creation of this data structure has to be
178  * done only once -  indicated by fact "unmerged" present
179  * in  case not performed before
180  */
181
182
183 name mergeInfo = "mergeInfo"
184 name state_control = "state_control"
185 name process = "process"
186
187 slot minNode = {PST_node unique_id
188     <- eval ({this flag} == 0)}
189
190 slot minNodeSink = {PST_node sink
191     <- eval ({this unique_id} == {mergeInfo cur_id})}
192
193 slot minNodeFlag = {PST_node flag
194     <- eval ({this unique_id} == {mergeInfo cur_id})}
195
196 slot minNodeNeighbor = {PST_node neighbor_id
197     <- eval ({this unique_id} == {mergeInfo cur_id})}
198
199 slot minNodeCost = {PST_node cost
200     <- eval ({this unique_id} == {mergeInfo cur_id})}
201
202 slot matchingNodeSink = {PST_node sink
203     <- eval ({this flag} == 0)
204     <- eval ({this sink} != minNodeSink)
205     <- eval ({this neighbor_id} == minNodeNeighbor)}
206
```

```
207 slot matchingNodeCost = {PST_node cost
208     <- eval ({this flag} == 0)
209     <- eval ({this sink} != minNodeSink)
210     <- eval ({this neighbor_id} == minNodeNeighbor)}
211
212 slot matchingNodeFlag = {PST_node flag
213     <- eval ({this flag} == 0)
214     <- eval ({this sink} != minNodeSink)
215     <- eval ({this neighbor_id} == minNodeNeighbor)}
216
217
218 /* The flag is used for traversing the datastructure.
219  * It is set to 1 for all PST_nodes that cannot be merged
220  */
221
222 name target = "target"
223
224
225 rule tagUnmergable 220
226 <- exists {data}
227 <- exists {unmergedYet}
228 -> retract {timer_context}
229 -> set {PST_node flag
230         <- eval ({this neighbor_id} != {PST_node neighbor_id})} =
              1
231 -> define state_control [state = "pst"]
232 -> define cachedMinCostSink [sink = 3, cost = 255]
233 -> flush {data}
234
235
236 //in case the PSTs are merged, jump directly towards routing
237
238 rule triggerRoutingDirectly 219
239 <- exists {data}
240 -> define process
241 -> flush {data}
242
243 rule startMerging 218
244 <- exists {state_control
245     <- eval ({this state} == "pst")}
246 <- exists {PST_node
247     <- eval ({this flag} == 0)}
248 -> define mergeInfo [cur_id = minNode, sink = 0, cost =
        minNodeCost]
249 -> set {mergeInfo sink} = minNodeSink
250 -> set {mergeInfo cost} = (minNodeCost - 1)
251 -> set minNodeFlag = 1
252 -> define PST_node [unique_id = pstNodeId, sink = (minNodeSink |
        matchingNodeSink), neighbor_id = minNodeNeighbor, cost = ({
        mergeInfo cost} + matchingNodeCost), flag = 0]
253 -> set matchingNodeFlag = 1
254 -> set pstNodeId = (pstNodeId + 1)
255 -> retract {mergeInfo}
256 -> set {state_control state} = "pst"
```

```
257
258
259 /* when all PST_nodes have been created merging is finished!
260  * retract the unmergedYet fact to indicate that the next
261  * incomingdata packet will not trigger merging again, but
262  * solely route construction
263  */
264
265 name broute = "broute"
266 name state = "state"
267 name croute = "croute"
268
269
270 rule mergingDone 217
271 <- exists {state_control
272     <- eval({this state} == "pst")}
273 <- eval((count {PST_node
274     <- eval ({this flag} == 0)}) == 0)
275 -> retract {unmergedYet}
276 -> set {cachedMinCostSink cost
277   <- eval ({this sink} == 3)} = (min {PST_node cost <- eval ({
        this sink} == 3)})
278 -> define broute [sink1 = 255, sink2 = 255, cost = 255, satisfies
      = 0]
279 -> define target [route = 0]
280 -> define state [current = "undef"]
281 -> retract {state_control}
282 -> set {data sink1} = {data sink1}
283
284
285 /*
286  * Phase 3 - all routing information is there: FORWARDING
287  * data can now be forwarded either via exploration or
288  * via exploitation. Incoming data may also feature feedback
289  * information that is used to update, the
290  * infrastructure, thus learn about the topology
291  */
292
293 /* --- CASE: FEEDBACK ---
294  * incoming data is feedback on the actual cost for a chosen
295  * route. the node is overhearing a paket targeting other nodes
296  * for forwarding, but can utilize the piggybacked  information
297  * for checking for new paths
298  */
299
300 //-- feed_cache [unique_id(PST), neighbor_id (who?)] --
301
302 name filterID = "filterID"
303 name feed_cache = "feed_cache"
304
305 slot filterID = {feed_cache unique_id
306     <- eval({this neighbor_id} == {data from})}
307
308
```

```
309 rule applyFeedback 216
310 <- exists {process}
311 <- exists {data
312     <- eval ({this from} == {feed_cache neighbor_id})}
313 -> define tmp [cur = {PST_node unique_id <- eval ({this unique_id
        } == filterID)}]
314
315 rule evalFeedbackBetter 215
316 <- exists {process}
317 <- exists {data
318     <- eval ({this from} == {feed_cache neighbor_id})}
319 <- eval ({PST_node cost
320   <- eval ({this unique_id} == {tmp cur})} > {data feedback})
321 -> set {PST_node cost
322   <- eval ({this unique_id} == {tmp cur})} = ({data feedback} +
        1)
323
324
325 rule learnedBetterPath 214
326 <- exists {process}
327 <- exists {data
328     <- eval ({this from} == {feed_cache neighbor_id})}
329 <- eval ({cachedMinCostSink cost
330   <- eval ({this sink} == {PST_node sink
331       <- eval ({this unique_id} == {tmp cur})})} > (min {PST_node
            cost
332                   <- eval ({this unique_id} == {tmp cur})}))
333 -> retract {broute}
334 -> define broute [sink1 = 255, sink2 = 255, cost = 255, satisfies
       = 0]
335 -> set {cachedMinCostSink cost
336   <- eval ({this sink} == {PST_node sink
337       <- eval ({this unique_id} == {tmp cur})})} = (min {PST_node
            cost
338                   <- eval ({this unique_id} == {tmp cur})})
339
340
341
342 rule cleanupFeedback 213
343 <- exists {process}
344 <- exists {data
345     <- eval ({this from} == {feed_cache neighbor_id})}
346 -> retract filterID
347 -> retract {tmp}
348 -> retract {data}
349 -> retract {process}
350
351
352 rule dropBouncingPakets 212
353 <- exists {process}
354 <- exists {data
355   <- eval ({this pkt_id} <= {system pkt_id})}
356 -> retract {data}
357 -> retract {process}
```

```
358
359 /* --- CASE: DATA HANDLING AT SINKS ---
360  * sinks count the acutal pakets that have arrived
361  * to later on derive the PDR and handle forwarding of
362  * those pakets that are heading towards the other sink
363  */
364
365
366 rule countPacketsAtSink1 201
367 <- exists {process}
368 <- exists {data
369   <- eval ({this sink1} == systemID)}
370 <- eval (systemID == 1)
371 -> set {stats current} = {data pkt_id}
372 -> set {stats received} = ({stats received} + 1)
373 -> set {data sink1} = 255
374 -> flush {data}
375
376
377 rule countPacketsAtSink2 200
378 <- exists {process}
379 <- exists {data
380   <- eval ({this sink2} == systemID)}
381 <- eval (systemID == 2)
382 -> set {stats current} = {data pkt_id}
383 -> set {stats received} = ({stats received} + 1)
384 -> set {data sink2} = 255
385 -> flush {data}
386
387
388 rule sendFeedbackSink 199
389 <- exists {process}
390 <- eval (systemID < 3)
391 <- exists {data
392   <- eval ({this sink2} == 255)
393   <- eval ({this sink1} == 255)}
394 -> set {data feedback} = 0
395 -> set {data from} = systemID
396 -> send systemBroadcast systemTxRange {data}
397 -> set {target route} = 0
398
399
400 rule getRidOfUnwantedData 198
401 <- exists {process}
402 <- exists {data
403   <- eval ({this sink1} != systemID)
404   <- eval ({this sink2} != systemID)}
405 <- eval ({state current} != "search")
406 -> retract{process}
407 -> retract {data}
408
409
410
411
```

```
412 /* --- CASE: GENERAL ROUTING ---
413  * first, set were the data is supposed to be
414  * forwarded to, then determine the mode (exploit or explore)
415  */
416
417
418 rule setTargetNodes1 180
419 <- exists {process}
420 <- exists {data
421     <- eval ({this sink1} == systemID)}
422 -> set {target route} = ({target route} | 1)
423 -> flush {target}
424
425
426 rule setTargetNodes2 179
427 <- exists {process}
428 <- exists {data
429     <- eval ({this sink2} == systemID)}
430 -> set {target route} = ({target route} | 2)
431 -> flush {target}
432
433
434 name explore = "explore"
435
436
437 rule determineRoutingModeExplore 177
438 <- exists {process}
439 <- exists {data}
440 <- eval (({system counter} modulo 3) == 2)
441 -> define explore
442 -> call getRandom
443 -> set {system counter} = ({system counter} + 1)
444 -> set {system counter} = ({system counter} modulo 10)
445 -> retract {process}
446 -> flush {data}
447
448
449 rule determineRoutingModeExploit 176
450 <- exists {process}
451 <- exists {data}
452 <- eval (({system counter} modulo 3) != 2)
453 -> set {state current} = "exploit"
454 -> set {system counter} = ({system counter} + 1)
455 -> set {system counter} = ({system counter} modulo 10)
456 -> define croute [cost = 0, satisfies = 0, ptr = 255]
457 -> retract {process}
458 -> flush {croute}
459
460
461 /* --- CASE: EXPLOIT ---
462  * this means, choose the best available route to the sink(s)
463  * and will be done in 30 percent  of the cases
464  */
465
```

```
466 //------ broute [sink1, sink2, cost, satisfies] ------
467 //----- croute [cost, satisfies, ptr] ------
468
469
470  rule prepareDataForRouting 175
471 <- exists {state
472     <- eval ({this current} == "exploit")}
473 -> set {system pkt_id} = {data pkt_id}
474 -> set {data sink1} = 255       //urx: hack.
475 -> set {data sink2} = 255
476 -> set {data feedback} = {cachedMinCostSink cost <- eval ({this
        sink} == {target route})}
477 -> set {data from} = systemID
478
479
480 name minMatchCost = "minMatchCost"
481 name minMatchID = "minMatchId"
482 name minMatchflag = "minMatchFlag"
483 name minMatchNB  = "minMatchNB"
484 name router = "router"
485
486 slot minMatchID = {PST_node unique_id
487   <- eval ({this flag} == 5)}
488
489 slot minMatchCost = {PST_node cost
490   <- eval ({this flag} == 5)}
491
492 slot minMatchFlag = {PST_node flag
493   <- eval ({this sink} == {target route})
494   <- eval ({this cost} == (min {PST_node cost
495                 <- eval({this sink} == {target route})}))}
496
497 slot minMatchNB = {PST_node neighbor_id
498   <- eval ({this flag} == 5)}
499
500 slot router = {PST_node neighbor_id
501   <- eval ({this unique_id} == {broute sink2})}
502
503
504 rule shortcutBrouteAvailable 174
505 <- exists {state
506     <- eval ({this current} == "exploit")}
507 <- eval({target route} == {broute satisfies})
508 -> set {data sink1} = {PST_node neighbor_id
509   <- eval ({this unique_id} == {broute sink1})}
510 -> set {data sink2} = {PST_node neighbor_id
511   <- eval ({this unique_id} == {broute sink2})}
512 -> send systemBroadcast systemTxRange {data}
513 -> set {state current} = "feedback"
514 -> set {target route} = 0
515 -> retract {croute}
516 -> retract {data}
517
518
```

```
519 rule findMinDirectly 160
520 <- exists {state
521     <- eval ({this current} == "exploit")}
522 <- exists {PST_node unique_id
523   <- eval ({this sink} == {target route})}
524 -> set {PST_node flag} = 1
525 -> set minMatchFlag = 5
526 -> set {broute cost} = minMatchCost
527 -> set {broute satisfies} = {target route}
528 -> set {broute sink1} = minMatchID
529 -> set {broute sink2} = minMatchID
530 -> set {data sink1} = minMatchNB
531 -> set {data sink2} = minMatchNB
532
533 rule correctSink1 159
534 <- exists {state
535     <- eval ({this current} == "exploit")}
536 <- eval ({target route} == 1)
537 -> set {broute sink2} = 255
538 -> set {data sink2} = 255
539 -> set {broute satisfies} = 1
540 -> define feed_cache [unique_id = {broute sink1}, neighbor_id =
        router]
541
542 rule correctSink2 158
543 <- exists {state
544     <- eval ({this current} == "exploit")}
545 <- eval ({target route} == 2)
546 -> set {broute sink1} = 255
547 -> set {data sink1} = 255
548 -> set {broute satisfies} = 2
549 -> define feed_cache [unique_id = {broute sink2}, neighbor_id =
        router]
550
551 rule sendDataSimple 157
552 <- exists {state
553     <- eval ({this current} == "exploit")}
554 <- eval ({target route} != 3)
555 -> send systemBroadcast systemTxRange {data}
556 -> set {cachedMinCostSink cost
557   <- eval ({this sink} == {target route})} = minMatchCost
558 -> set {target route} = 0
559 -> set {state current} = "undef"
560 -> set {PST_node flag} = 1
561 -> retract {croute}
562 -> retract {data}
563
564 rule setStateToSearchRoute 156
565 <- exists {state
566     <- eval ({this current} == "exploit")}
567 -> flush {data}
568 -> set {state current} = "search"
569
570
```

```
571 /*
572  * search for possible better matches than the ones
573  * denoted directly in the PST_nodes by constructing
574  * all possible routes (croute) and checking them agains the
575  * current best route (broute)
576  */
577
578 name startNode = "startNode"
579
580 slot startNode = {PST_node unique_id
581     <- eval ({this flag} != 2)}
582
583 slot startNodeSink = {PST_node sink
584     <- eval ({this unique_id} == {mergeInfo cur_id})}
585
586 slot startNodeNeighbor = {PST_node neighbor_id
587     <- eval ({this unique_id} == {mergeInfo cur_id})}
588
589 slot startNodeCost = {PST_node cost
590     <- eval ({this unique_id} == {mergeInfo cur_id})}
591
592 slot potentialNodeSink = {PST_node sink
593     <- eval ({this flag} == 1)
594     <- eval ({this sink} != startNodeSink)
595     <- eval ({this neighbor_id} != startNodeNeighbor)}
596
597 slot potentialNodeID = {PST_node unique_id
598     <- eval ({this flag} == 1)
599     <- eval ({this sink} != startNodeSink)
600     <- eval ({this neighbor_id} != startNodeNeighbor)}
601
602 slot potentialNodeCost = {PST_node cost
603     <- eval ({this flag} == 1)
604     <- eval ({this sink} != startNodeSink)
605     <- eval ({this neighbor_id} != startNodeNeighbor)}
606
607 slot potentialNodeFlag = {PST_node flag
608     <- eval ({this flag} == 1)
609     <- eval ({this sink} != startNodeSink)
610     <- eval ({this neighbor_id} != startNodeNeighbor)}
611
612
613 rule startSearching 150
614 <- exists {state
615     <- eval ({this current} == "search")}
616 <- exists {PST_node
617     <- eval ({this flag} == 1)}
618 <- eval ((count {PST_node
619         <- eval ({this flag} == 10)}) == 0)
620 -> define mergeInfo [cur_id = startNode, sink = 0, cost = 0]
621 -> set{mergeInfo sink} =  startNodeSink
622 -> set {mergeInfo cost} = (startNodeCost - 1)
623 -> set {PST_node flag
624   <- eval({this unique_id} == {mergeInfo cur_id})} = 10
```

```
625
626 rule isThereAMatch 149
627 <- exists {state
628     <- eval ({this current} == "search")}
629 <- exists {mergeInfo}
630 <- exists potentialNodeSink
631 -> set {croute cost} = ({mergeInfo cost} + potentialNodeCost)
632 -> set {croute ptr} = potentialNodeID
633 -> set {croute satisfies} = (potentialNodeSink | {mergeInfo sink
    })
634 -> set potentialNodeFlag = 11
635
636
637 rule isMatchBetterThanBroute 148
638 <- exists {state
639     <- eval ({this current} == "search")}
640 <- exists {mergeInfo}
641 <- eval ({croute satisfies} == {target route})
642 <- eval ({croute cost} < {broute cost})
643 -> set {broute cost} = {croute cost}
644 -> set {PST_node flag
645   <- eval({this flag} == 10)} = 42
646 -> set {PST_node flag
647   <- eval({this unique_id} == {croute ptr})} = 42
648 -> set {broute sink1} = {PST_node unique_id
649   <- eval ({this flag} == 42)
650   <- eval ({this sink} == 1)}
651 -> set {broute sink2} = {PST_node unique_id
652   <- eval ({this flag} == 42)
653   <- eval ({this sink} == 2)}
654 -> set {PST_node flag
655   <- eval({this unique_id} == {croute ptr})} = 11
656 -> set {PST_node flag
657   <- eval({this unique_id} == {mergeInfo cur_id})} = 10
658
659
660 rule restartLoop 147
661 <- exists {state
662     <- eval ({this current} == "search")}
663 <- exists potentialNodeSink
664 -> set {state current} = {state current}
665
666
667 rule clear 146
668 <- exists {state
669     <- eval ({this current} == "search")}
670 -> retract {mergeInfo}
671 -> set {PST_node flag
672   <- eval ({this flag} == 10)} = 2
673 -> set {PST_node flag
674   <- eval ({this flag} == 11)} = 1
675
676
677
```

```
678 rule finalRound 145
679 <- exists {state
680     <- eval ({this current} == "search")}
681 <- exists {PST_node flag
682   <- eval ({this flag} == 1)}
683 -> set {state current} = {state current}
684
685
686 rule sendDataAfterLoop 144
687 <- exists {state
688     <- eval ({this current} == "search")}
689 -> set {data sink1} = {PST_node neighbor_id
690         <- eval ({this unique_id} == {broute sink1})}
691 -> set {data sink2} = {PST_node neighbor_id
692         <- eval ({this unique_id} == {broute sink2})}
693 -> set {cachedMinCostSink cost
694   <- eval ({this sink} == {target route})} = {broute cost}
695 -> set {data feedback} = {cachedMinCostSink cost <- eval ({this
        sink} == {target route})}
696 -> set {data from} = systemID
697 -> send systemBroadcast systemTxRange {data}
698 -> set {target route} = 0
699 -> set {system pkt_id} = {data pkt_id}
700 -> set {PST_node flag
701   <- eval ({this flag} != 1)} = 1
702 -> set {state current} = "feedback"
703 -> retract {croute}
704 -> retract {data}
705
706 /*
707  * after successfully sending the data, the node stores
708  * the costs and the neighbor it has utilized within a fact
709  * to apply feedback on eventually smaller costs thereafter
710  */
711
712 name neighborInCache1 = "neighborInCache1"
713 name neighborInCache2 = "neighborInCache2"
714
715 slot neighborInCache1 = {PST_node neighbor_id
716         <- eval ({this unique_id} == {broute sink1})}
717
718 slot neighborInCache2 = {PST_node neighbor_id
719         <- eval ({this unique_id} == {broute sink2})}
720
721
722 rule provideFeedBack 140
723 <- exists {state
724   <- eval ({this current} == "feedback")}
725 <- eval (systemID > 2)
726 <- eval ((count {feed_cache
727           <- eval ({this unique_id} == {broute sink1})}) == 0)
728 <- eval ({broute sink1} != 255)
729 -> define feed_cache [unique_id = {broute sink1}, neighbor_id =
        neighborInCache1]
```

```
730
731 rule differentNeighbors 138
732 <- exists {state
733   <- eval ({this current} == "feedback")}
734 <- eval (systemID > 2)
735 <- eval((count {feed_cache
736         <- eval ({this unique_id} == {broute sink2})}) == 0)
737 <- eval ({broute sink1} != {broute sink2})
738 <- eval ({broute sink2} != 255)
739 -> define feed_cache [unique_id = {broute sink2}, neighbor_id =
        neighborInCache2]
740
741
742 rule finishedExploitation 136
743 <- exists {state
744   <- eval ({this current} == "feedback")}
745 -> set {state current} = "undef"
746
747
748 /*
749  *  --- CASE: EXPLORE ---
750  * when sent using the exploration strategy, a random value
751  * will be used to determine which route to pick
752  */
753
754 name random = "random"
755 name selected = "selected"
756
757 rule prepareDataForRoutingExplore 121
758 <- exists {explore}
759 -> set {system pkt_id} = {data pkt_id}
760 -> set {data sink1} = 255       //urx: hack. no match, value
        stays
761 -> set {data sink2} = 255
762 -> set {data feedback} = {cachedMinCostSink cost <- eval ({this
        sink} == {target route})}
763 -> set {data from} = systemID
764
765
766 rule findGoodExploration 120
767 <- exists {explore}
768 -> define tmp [cur = (count {PST_node})]
769 -> set {tmp cur} =  ({random value}  modulo {tmp cur})
770 -> set {tmp cur} = ({tmp cur} + 1)
771 -> define selected [ptr = 255]
772 -> set {selected ptr} = {PST_node unique_id
773         <- eval ({this sink} == {target route})
774         <- eval ({this unique_id} >= {tmp cur})}
775 -> define selected [ptr = 255]
776 -> set {selected ptr <- eval ({this ptr} == 255 )} = {PST_node
        unique_id
777         <- eval ({this sink} == {target route})
778         <- eval ({this unique_id} <= {tmp cur})}
779
```

```
780 rule checkOnePath 119
781 <- exists {explore}
782 <- eval ({selected ptr} != 255)
783 -> set {data sink1} = {PST_node neighbor_id
784          <- eval ({this unique_id} == {selected ptr})
785          <- eval ({this sink} == {target route})}
786 -> set {data sink2} = {PST_node neighbor_id
787          <- eval ({this unique_id} == {selected ptr})
788          <- eval ({this sink} == {target route})}
789 -> set {data feedback} = {PST_node cost
790          <- eval ({this unique_id} == {selected ptr})}
791 -> send systemBroadcast systemTxRange {data}
792 -> define feed_cache [unique_id = {selected ptr}, neighbor_id = {
    data sink1}]
793 -> set {target route} = 0
794 -> retract {explore}
795 -> retract {data}
796 -> retract {random}
797 -> retract {tmp}
798 -> retract {selected}
799 -> set {state current}  = "undef"
800
801
802
803 rule noSharedPath 118
804 <- exists {explore}
805 <- eval ({selected ptr} == 255)
806 -> set {data sink1} = {PST_node neighbor_id
807          <- eval ({this sink} == 1)}
808 -> set {data sink2} = {PST_node neighbor_id
809          <- eval ({this sink} == 2)}
810 -> send systemBroadcast systemTxRange {data}
811 -> set {target route} = 0
812 -> retract {explore}
813 -> retract {data}
814 -> retract {random}
815 -> retract {tmp}
816 -> retract {selected}
817 -> set {state current}  = "undef"
```

B.4 The Fence Monitoring ruleset

Listing B.4: The Fence Monitoring ruleset

```
1  ruleset FenceMonitoring
2
3  /*
4   * This ruleset implements fence monitoring, i.e. it aggregates
5   * low-level events (both locally and within an
6   * n-hop neighborhood) and routes high-level
7   * events to a base station.
8   */
9
10 name system = "system"
11 fact system [broadcast = 255, tx-range = 10]
12 slot systemID = {system owner}
13 slot systemBroadcast = {system broadcast}
14 slot systemTxRange = {system tx-range}
15
16 //Define standard names and slots for this ruleset.
17
18 name init = "init"
19 name shake = "shake"
20 name climb = "climb"
21 name alert = "alert"
22
23 /*
24  * Step 0: Build Routing Tree / Route Alerts to Sink
25  * Build a spanning tree for routing and send any
26  * alert facts back to the sink.
27  */
28
29 name createRoute = "createRoute"
30 name route = "route"
31 fact route [nextHop = 255]
32 slot routeNextHop = {route nextHop}
33
34
35 rule buildRoutingTreeOnInit 250
36 <- exists {init}
37 -> set routeNextHop = systemID
38 -> define createRoute [source = systemID]
39 -> send systemBroadcast systemTxRange {createRoute}
40 -> retract {createRoute}
41 -> retract {init}
42
43
44 rule addRoute 240
45 <- exists {createRoute}
46 <- eval (routeNextHop == 255)
47 -> set routeNextHop = {createRoute source}
48 -> set {createRoute source} = systemID
49 -> send systemBroadcast systemTxRange {createRoute}
50 -> retract {createRoute}
```

```
51
52 rule retractCreateRoute 235
53 <- exists {createRoute}
54 -> retract {createRoute}
55
56
57 rule processAlertsAtSink 230
58 <- exists {alert}
59 <- eval (routeNextHop == systemID)
60 -> call printFact ({alert})
61 -> retract {alert}
62
63
64 rule routeAlertsToSink 225
65 <- exists {alert}
66 -> send routeNextHop systemTxRange {alert}
67 -> retract {alert}
68
69
70 /*
71  * Step 1: Node-Local Event Processing
72  * Aggregate low-level shake events into a high-level climb event
73  */
74
75
76 name localAggregation = "localAggregation"
77 fact localAggregation [
78   discardEventsAfter = 30,
79   minShakeIntensity = 200,
80   minShakeDuration = 100,
81   minCombinedShakeDuration = 500,
82   maxCombinedShakeDuration = 1750,
83   minShakeEventsTrigger = 3
84 ]
85
86
87 slot localAggregationDiscardEventsAfter = {localAggregation
       discardEventsAfter}
88 slot localAggregationMinShakeIntensity = {localAggregation
       minShakeIntensity}
89 slot localAggregationMinShakeDuration= {localAggregation
       minShakeDuration}
90 slot localAggregationMinCombinedShakeDuration= {localAggregation
       minCombinedShakeDuration}
91 slot localAggregationMaxCombinedShakeDuration= {localAggregation
       maxCombinedShakeDuration}
92 slot localAggregationMinShakeEventsTrigger= {localAggregation
       minShakeEventsTrigger}
93
94
95 slot newShakeTime = {shake time
96   <- eval ({this modified} == true)}
97
98
```

```
99  rule purgeOldAndWeakShakeEvents 200
100 <- exists {shake}
101 -> retract {shake
102   <- eval ({this time} < (newShakeTime -
          localAggregationDiscardEventsAfter))}
103 -> retract {shake
104   <- eval ({this intensity} <= localAggregationMinShakeIntensity)
          }
105 -> retract {shake
106   <- eval ({this duration} <= localAggregationMinShakeDuration)}
107
108
109 rule aggregateShakeEvents 190
110 <- exists {shake}
111 <- eval ((count {shake}) >= localAggregationMinShakeEventsTrigger
       )
112 <- eval ((sum {shake duration}) >=
       localAggregationMinCombinedShakeDuration)
113 <- eval ((sum {shake duration}) <=
       localAggregationMaxCombinedShakeDuration)
114 -> define climb [confidence = ((max {shake intensity}) * (max {
       shake duration}))]
115 -> retract {shake}
116
117
118 /*
119  * Step 2: One-Hop Neighborhood Event Aggregation
120  * Broadcast new climb events to one-hop neighbors,
121  * reply with ACK or NACK depending on local events.
122  * After a timer runs out, decide whether to send
123  * the event to the base station.
124  */
125
126 name ack = "ack"
127 name nack = "nack"
128 name neighborhoodSendDelayTimerExpired = "
       neighborhoodSendDelayTimerExpired"
129 name neighborhoodAggregationTimerExpired = "
       neighborhoodAggregationTimerExpired"
130 name neighborhoodAggregation = "neighborhoodAggregation"
131
132
133 fact neighborhoodAggregation [
134   minShakeEventsTrigger = 3,
135   minAckTrigger = 1
136 ]
137
138
139 slot neighborhoodAggregationMinShakeEventsTrigger= {
       neighborhoodAggregation minShakeEventsTrigger}
140
141 slot neighborhoodAggregationMinAckTrigger= {
       neighborhoodAggregation minAckTrigger}
142
```

```
143 slot newReceivedClimbEvent = {climb
144   <- eval ({this modified} == true)
145   <- eval ({this owner} != systemID)}
146
147 slot newReceivedClimbEventID = {climb id
148   <- eval ({this modified} == true)
149   <- eval ({this owner} != systemID)}
150
151 slot newReceivedClimbEventOwner = {climb owner
152   <- eval ({this modified} == true)
153   <- eval ({this owner} != systemID)}
154
155
156 rule delayOnNewLocalClimbEvents 150
157 <- exists {climb
158   <- eval ({this owner} == systemID)}
159 -> call setTimer ({neighborhoodSendDelayTimerExpired}, 1)
160
161 rule broadcastNewLocalClimbEvents 145
162 <- exists {neighborhoodSendDelayTimerExpired}
163 -> send systemBroadcast systemTxRange {climb}
164 -> retract {neighborhoodSendDelayTimerExpired}
165 -> call setTimer({neighborhoodAggregationTimerExpired}, 3)
166
167 rule ackNewReceivedClimbEvents1 140
168 <- exists newReceivedClimbEvent
169 <- exists {climb
170   <- eval ({this owner} == systemID)}
171 -> define ack [eventID = newReceivedClimbEventID]
172 -> send newReceivedClimbEventOwner systemTxRange {ack
173   <- eval ({this owner} == systemID)}
174 -> retract {ack
175   <- eval ({this owner} == systemID)}
176 -> retract newReceivedClimbEvent
177
178 rule ackNewReceivedClimbEvents2 130
179 <- exists newReceivedClimbEvent
180 <- eval ((count {shake}) >=
      neighborhoodAggregationMinShakeEventsTrigger)
181 -> define ack
182 -> send newReceivedClimbEventOwner systemTxRange {ack
183   <- eval ({this owner} == systemID)}
184 -> retract {ack
185   <- eval ({this owner} == systemID)}
186 -> retract newReceivedClimbEvent
187
188 rule nackNewReceivedClimbEvents 120
189 <- exists newReceivedClimbEvent
190 -> define nack [eventID = newReceivedClimbEventID]
191 -> send newReceivedClimbEventOwner systemTxRange {nack
192   <- eval ({this owner} == systemID)}
193 -> retract {nack
194   <- eval ({this owner} == systemID)}
195 -> retract newReceivedClimbEvent
```

```
196
197 rule evalAcksOnTimeout 110
198 <- exists {neighborhoodAggregationTimerExpired}
199 <- eval ((count {ack}) >= neighborhoodAggregationMinAckTrigger)
200 -> define alert [confidence = {climb confidence}]
201
202 rule retractAcksAfterTimeout 100
203 <- exists {neighborhoodAggregationTimerExpired}
204 -> retract {ack}
205 -> retract {neighborhoodAggregationTimerExpired}
206 -> retract {climb}
```

Appendix C

Zusammenfassung

Die fortschreitende Miniaturisierung technischer Bauteile erlaubt mittlerweile den flächendeckenden Einsatz kleinster Rechner, die sich durch drahtlose Kommunikation miteinander verbinden. Ausgestattet mit einer Vielzahl von Sensoren finden sich diese sogenannten drahtlosen Sensorknoten in ad-hoc Netzen zusammen, und ermöglichen so eine Vielzahl neuartiger Anwendungen. Die Programmierung dieser Sensornetze ist allerdings komplex und sehr fehleranfällig, da viele Faktoren wie die räumliche Verteilung, die unzuverlässige drahtlose Kommunikation und die Programmierung eingebetteter Systeme berücksichtigt werden müssen.

Diese Dissertation stellt eine regelbasierte, domänen-spezifische Sprache und ein dazugehöriges Rahmenwerk vor, welches dem Programmierer eine abstrakte, problem-orientierte Sichtweise auf das Sensornetz zur Verfügung stellt. Neu ist, dass der Programmierer mit präzisen, mächtigen Sprachelementen die Reaktionen eines Sensorknotens auf komplexe Ereignisse definieren kann, ohne sich systembedingter Abläufe bewusst sein zu müssen. Verschiedene Optimierungsverfahren wurden vorgestellt, die sowohl die Laufzeit des Systems beschleunigen, als auch den Speicherverbrauch minimieren. Die Evaluation hat nicht nur die Qualität des Ansatzes in unterschiedlichsten Szenarien unter Beweis gestellt, sondern auch quantitativ nachgewiesen, dass die durch die Abstraktion bedingten, durchschnittlichen Verluste in Reaktivität keinen signifikanten Einfluss auf seine Nutzbarkeit haben.

Appendix D

Erklärung

Ich versichere, dass ich die vorliegende Dissertation auf Grundlage der in der Arbeit angegebenen Hilfsmittel und Hilfen selbständig verfasst habe.

Bern, den 5. Mai 2009

Kirsten Terfloth