FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Towards a Toolchain for Asynchronous Embedded Programming based on the Peer-Model

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Thomas Hamböck
Matrikelnummer 0828408

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: A.o.Univ.-Prof. Dr. Dipl.-Ing. eva Kühn
Mitwirkung: Dipl.-Ing. Stefan Craß

Wien, 19. April 2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Towards a Toolchain for Asynchronous Embedded Programming based on the Peer-Model

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

by

Thomas Hamböck
Registration Number 0828408

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: A.o.Univ.-Prof. Dr. Dipl.-Ing. eva Kühn
Assistance: Dipl.-Ing. Stefan Craß

Vienna, 19. April 2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Thomas Hamböck
Graf-Starhemberg-Gasse 5 / 1 / 30-31, 1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)

(Unterschrift Verfasser)

Acknowledgements

Hereby, I want to express my appreciation to my advisers Prof. Dr. eva Kühn and Dipl.-Ing. Stefan Craß for their support throughout the thesis and the time I was involved in various research projects.

Furthermore, I would like to thank the members of the Space Based Computing Group and other colleagues who gave me a lot of feedback Thanks, in alphabetical order, are especially directed to Michael Borko, Stephan Cejka, Matthias Fassl, Georg Holasek, Stefan Krulj and Matthias Schwayer.

Additionally I want to thank my girlfriend Sabrina for her mental support and faith in me during my studies and especially in the final phase of my master studies. I am also grateful for the motivation provided by Marion, a good friend of mine.

Finally, special thanks have to be given to my father for the financial support during my studies and even more for introducing me into the practice of electrical engineering and showing me the basics of microcontroller programming when I was young.

Abstract

In recent years, the field of wireless sensor networks grew more and more. Suitable wireless sensor nodes are getting cheaper and cheaper and therefore, even large quantities are affordable for amateur home automation and other smaller projects. Nevertheless, toolchain support and software development principles are far from optimal. The focus of this work is to bring software engineering methods to embedded systems programming and the development of a toolchain for distributed embedded systems, supporting embedded software engineers in design and implementation.

The motivating use case for this work comes from the railway telematics domain where copper cables shall be replaced by wireless smart nodes along a railway track. A wheel sensor beside the track reports approaching trains and this information has to be routed over the wireless network to a controller at the level crossing.

The Peer Model is a programming model especially aiming to improve the development of coordination for distributed and concurrent systems. It is based on a space-based abstraction and uses an asynchronous, event-driven approach. This model is therefore adapted to fit wireless sensor networks' needs. Then a domain specific language is developed which shall serve as the basis for a holistic toolchain. To bridge the gap between coordination design and embedded implementation a compiler generating ANSI C code for embedded platforms is implemented. The ANSI C framework is implemented for one Arduino based controller and one wireless sensor node optimised with respect to energy efficiency.

Finally the advantages of the developed toolchain are evaluated by implementing different case studies. The implementations of the scenarios are benchmarked regarding framework overhead, energy efficiency and source code changes implied by changing requirements. For the motivating use case also fieldtests beside a railway track are carried out and analysed.

Kurzfassung

In den letzten Jahren setzen sich drahtlose Sensornetzwerke mehr durch und entsprechende Funkknoten werden immer günstiger. Dies ermöglicht auch den Einsatz von Sensornetzwerken für kleinere Projekte oder den Privatgebrauch. Nichtsdestotrotz ist die Toolchain Unterstützung für die Softwareentwicklung noch alles andere als optimal. Das Ziel dieser Arbeit liegt darin, Methodik und Unterstützung in Form einer Toolchain in die Programmierung von verteilten, eingebetteten Systemen zu bringen.

Die zugrundeliegende Anwendung stammt aus dem Eisenbahnbereich, bei dem Kupferkabel entlang einer Eisenbahnstrecke durch intelligente Funkknoten ersetzt werden sollen. Hier soll das Signal eines Radsensors, der ankommende Züge erkennt, entlang der Strecke verlässlich zur Eisenbahnkreuzung übertragen werden.

Das Peer Modell ist ein Programmiermodell welches die Entwicklung von Koordinationssoftware für verteilte und parallele Systeme vereinfacht. Es basiert auf Prinzipien von “shared spaces” und folgt einem asynchronen, ereignisgesteuerten Ansatz. Dieses Modell wird an die Anforderungen von drahtlosen Sensornetzwerke angepasst und eine domänenspezifische Sprache entwickelt, die als Grundlage für eine umfassende Toolchain dienen soll. Um die Programmierung zu vereinfachen wird außerdem ein Übersetzer für eine ANSI C Implementierung für eingebettete Systeme realisiert, der eine Arduino-basierte und eine eigens entwickelte energieoptimierte Plattform unterstützt.

Abschließend werden die Vorteile der entwickelten Toolchain durch die Umsetzung verschiedener Einsatzszenarien, unter anderem der Eisenbahnanwendung, hinsichtlich dem benötigten Speicherplatz, der Energieeffizienz und der durch veränderte Anforderungen hervorgerufenen benötigen Änderungen im Quellcode analysiert. Für die Anwendung im Eisenbahnbereich werden außerdem Feldtests an einer Eisenbahnstrecke durchgeführt.

Contents

List of Figures

CHAPTER 1

Introduction

Since several years Wireless Sensor Networks (WSNs) are on the rise and also appear more and more in the field of amateur home automation. For non-military applications, wireless sensor nodes built of an Arduino-based controller and a ZigBee transceiver are well-established. In recent years also Micro Controller Units (MCUs) and wireless radios with lower a energy consumption have been developed enabling the operation of battery powered devices for longer periods of time.

Beside the widespread Arduino community, where the framework only provides a Hardware Abstraction Layer (HAL) and abstraction only comes from a class-based C++ design, a first real level of abstraction is introduced by using an Operating System (OS) tailored to the needs of embedded hardware, like TinyOS [47], Contiki [18] or RIOT OS [3]. These OSs typically offer a basic hardware abstraction for generic peripherals, task scheduling mechanisms and/or simple memory management to application developers.

Especially in the field of distributed embedded systems a higher level of abstraction would be desired to keep a clear view and retain control of highly concurrent systems. However, developers of embedded software (in non-safety critical domains) often tend to steer clear of traditional software development principles. Usually the main reason for this issue is missing tool support. Typically development and debugging tools for embedded systems, like Integrated Development Environments (IDEs), Joint Test Action Group (JTAG) debuggers and JTAG tracers, are very costly and therefore not affordable for smaller developments. Simulation and verification tools are very uncommon as well because external peripherals are usually hard to simulate as hardware descriptions are incomplete or even missing.

This work focuses on the programming of highly concurrent automated Wireless Sensor and Actor Networks (WSANs) where no central coordinator and only peer-to-peer network connections are assumed. Tuple space-based coordination [34] is a data-driven approach where coordination is carried out via shared spaces and offers decoupling in time and space. Since a space-based coordination approach has been proven applicable and showed great benefits in the WSAN domain, for example in [9], a toolchain based on those principles shall be developed to

support developers in the design and realisation process of applications requiring coordination as in the distributed embedded systems domain.

The proposed programming model, which has first been introduced in [46] and got refined in [45], builds rule-based coordination mechanisms on top of a space abstraction called XVSM [11, 44]. Since embedded systems have a lower computational power and less memory than conventional desktop computers and typically miss useful features like dynamic memory management, multi-threading and high-speed network connections, the programming model has to be adapted to meet those requirements.

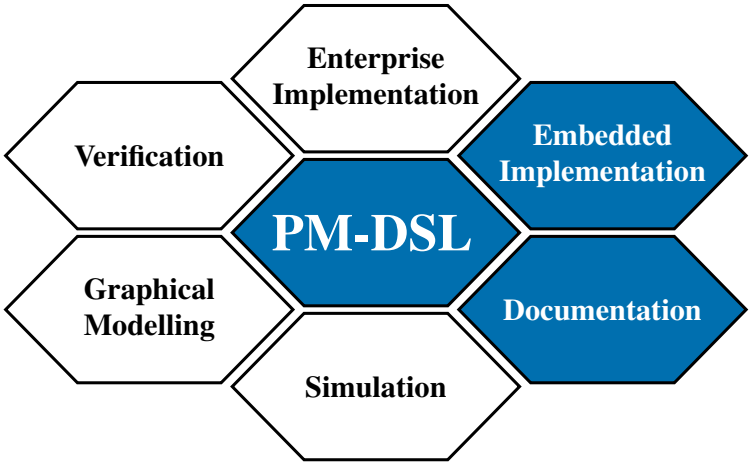


Figure 1.1: Overview of the Peer Model Toolchain. Blue parts are covered in this thesis and white ones are planned or work in progress.

Furthermore, to enable the development of a holistic toolchain a Domain Specific Language (DSL), representing the centre of a toolchain, shall be defined. The interplay of the components of the proposed toolchain is illustrated in Figure 1.1. Since documentation plays an important (but sadly often neglected) role in software development, a graphical documentation support shall be given as well. This documentation generation aims at software reviewing sessions, the reconstruction of existing software pieces and the analysis of possible software compositions. Finally, to eliminate mistakes which arise from implementing a specified application behaviour in a programming language, an automatic code generation is crucial.

The space-based coordination abstraction in combination with tool support should improve software development for automated WSANs. The motivating use case for the development of the coordination abstraction and toolchain development, originating from research projects, is introduced in Section 1.1.

During the work on the thesis, intermediate results of Chapter 4 and Chapter 6 have been published in [42].

This work was partially funded by:

Project "LOPONODE-Middleware" The project "LOPONODE-Middleware" was funded under the programme "FFG BRIDGE" by the Austrian Federal Ministry for Transport, Innovation and Technology (bmvit) with the project number 834162. The development of the DSL (Chapter 4), the reference implementation of the Embedded Peer Model (Chapter 6) and the fieldtests (Section 7.3) have been part of this project.

Project "LOPONODE Proof-of-Concept" The development of energy-optimised LOPONODE hardware, as shown in Section 5.4, has been part of the project "LOPONODE Proof-of-Concept", funded by the industrial partner "ÖBB Infrastruktur AG"[1].

1.1 Motivating Use Case

The motivating use case has first been introduced in [46, 45] and comes from the railway telematics domain. Approaching trains are recognised by a wheel sensor mounted beside the track (the yellow box in Figure 1.2) and this information is sent to the Level Crossing Controller (LCC) (the big grey box). Then the signal lights at the level crossing are activated which means that cars are not allowed to traverse the level crossing anymore. Afterwards the LCC activates a train notification light (the pole in the middle) signalling the train that it is allowed to pass the crossing. In case the train notification signal is not activated, e.g. when the wheel sensor or the LCC has a failure, the engine driver has to reduce the train's speed and blow the horn before passing the level crossing.

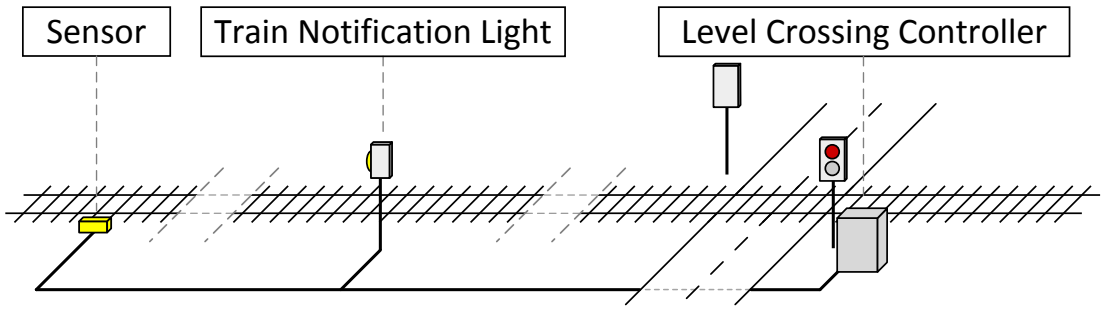


Figure 1.2: Railway Use Case: Traditional Copper Cables.

Traditionally the interconnections of the LCC, the wheel sensor and the train notification light are made with copper cables. This conventional realisation is sketched in Figure 1.2. Since the copper prices rose in recent years, lots of copper cables were stolen from level crossings and sold elsewhere. To cope with copper cable theft and to reduce cabling costs for new level

[1] `http://www.oebb.at/infrastruktur/de/`

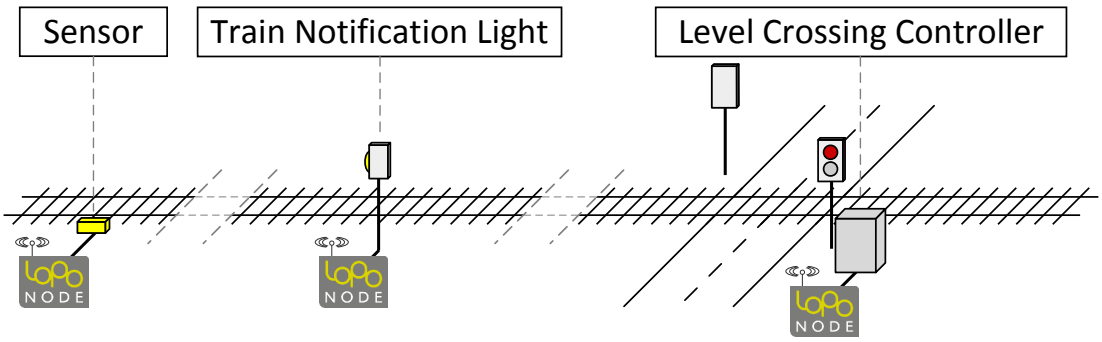


Figure 1.3: Railway Use Case: Wireless Signal Transmission.

crossings, these wires shall be replaced by a wireless link. The usage of a special energy-optimised hardware, called Low-Power Node (LOPONODE), comprising a MCU, a wireless radio and energy harvesting mechanisms, is contemplated. A scenario where copper cables are replaced by self-sustaining LOPONODEs is depicted in Figure 1.3.

1.2 Content and Structure of the Thesis

First, the state of the art survey for subsequent chapters is given in Chapter 2. Basic terminology of wireless sensor networks is covered in Section 2.1, and existing wireless sensor network hardware is presented in Section 2.2. The concepts of space-based computing are described in Section 2.3, then an insight into DSL development is given in Section 2.4 and finally related work is identified and covered in Section 2.5.

Chapter 3 comprises an overview of the Peer Model, the programming model which is used in this thesis and is based on the concepts of space-based computing. For the embedded case some restrictions have to be made on the traditional Peer Model which is then called the Embedded Peer Model (ePM).

To bridge the gap between the Peer Model specification, executable code and documentation generation, a DSL for the ePM is introduced in Chapter 4. This DSL plays the central role in the Peer Model toolchain, outlined in Figure 1.1. At first, design decisions are stated, then the DSL is specified formally and source code examples are shown.

Since this work targets the realisation of the motivating use case described in Section 1.1, used hardware plays a central role. Therefore an outline of selected hardware is given in Chapter 5. At the beginning, Section 5.1 states the requirements which have to be fulfilled for the hardware being used. Then, Section 5.3 presents a Commercial Off The Shelf (COTS) wireless sensor network node, demonstrating the first prototype implementation. In Section 5.4 the node developed and used during the research projects is described.

Chapter 6 gives attention to the reference implementation of the Peer Model Compiler, shown in Section 6.1, which transforms the DSL to different output targets. These targets are represented by the blue outer honeycombs in Figure 1.1, for example the ANSI C framework, described in Section 6.2, and the LaTeX documentation generation, presented in Section 6.3.

In Chapter 7, the ePM reference implementation is evaluated on the basis of different case studies. The comparison regarding source code size, binary size and energy performance is carried out by implementing some scenarios with the native Arduino framework and the ePM implementation for the Arduino platform and the LOPONODEs. Then more complex scenarios are implemented with the Peer Model DSL and composition and re-usability is analysed. Also an estimation of the energy consumption and performance of both node types is given. At the end of the chapter, the ePM is compared to related space-based embedded middlewares.

Finally an overview over future work is given in Section 8 and the work is concluded in Chapter 9.

CHAPTER 2

State of the Art

2.1 Wireless Sensor Networks (WSNs)

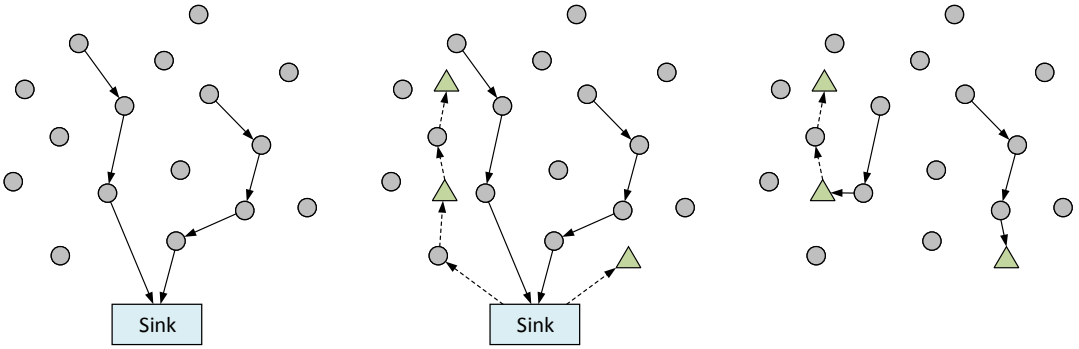
WSNs [67] consist of distributed embedded nodes, also called "motes", which are able to sense physical and environmental quantities, process gathered values and forward them to a sink node. The sink node is usually a Personal Computer (PC) which records and triggers actions based on sensed data. An example for a WSN is illustrated in Figure 2.1a. Typical applications of WSNs are monitoring of animals, traffic, factories or the environment.

WSANs [1] are similar to WSNs but also include actor nodes which are able to modify their surrounding environment. WSANs are further classified in automated and semi-automated WSANs [1]. Semi-automated WSANs are characterised by a sink node where all sensor data gets aggregated and actor nodes are controlled from. A typical setup of a semi-automated WSAN is shown in Figure 2.1b. For this kind of network only sensor-sink and sink-actor communication is necessary. Automated WSANs are able to to operate without a sink node and therefore mainly sensor-actor and actor-actor communication is used. An example for an automated WSAN is depicted in Figure 2.1c. Please note that sensor and actor nodes can be mixed to so-called sensor/actor-nodes.

Advantages and Disadvantages of Automated WSANs

Advantages of automated, sinkless WSANs in contrast to WSNs and semi-automated WSANs, according to [1], are:

Latency Automated WSAN have the advantage that distances between sensors and associated actors are shorter than the sensor-sink-actor distance and can therefore operate more efficiently.



(a) WSN example. (b) Semi-automated WSAN. (c) Automated WSAN.

Figure 2.1: Typical WSN and WSAN Scenarios. Circles denote sensor nodes and triangles denote actor nodes.

Energy Consumption WSANs without a sink node have the advantage that they will not require as many forwarder (relay) nodes to convey messages to corresponding receiver nodes which saves energy over the whole network.

Network Lifetime In WSNs and semi-automated WSANs, illustrated in Figure 2.1b, nodes near the sink are more utilised than nodes at the border of the network. When nodes near the sink run out of energy, the network may not be able to operate successfully anymore. For automated WSANs, shown in Figure 2.1c, each node's utilisation should be nearly uniformly distributed, thus increasing the network lifetime.

Complexity In contrast to semi-automated WSANs, automated WSANs are more complex with respect to software development and debugging. For the semi-automated case, software updates, debugging and logging can be done on the sink node.

Although the considered use case may be solved with a semi-automated WSAN in this scenario, this may not be true anymore if hardware redundancy (e.g. by using multiple chains) is demanded. Therefore the focus of this work is only given to automated WSANs without a sink node.

Challenges of WSANs

According to [1], the main challenges of WSANs are:

Node Heterogeneity Sensor nodes generally consume less energy than actor nodes. Since actor nodes have a higher Central Processing Unit (CPU) usage because they need to process sensed data and attached actuators also have to be supplied with energy to control them, these nodes are typically built different.

For our use case introduced in Section 1.1, attached actuators are supplied independently and the actuators' power supply wire may not be tapped and sensed data is assumed to be prepro-

cessed, reducing the CPU load on actor nodes. Therefore all nodes are realised as self-sustaining sensor/actor nodes.

Real-time Requirements In some WSANs, depending on the use case, the timing of data transmissions may be constrained and real-time requirements have to be met.

Our use case does not require real-time constraints. In the case of losing wheel sensor information or too long transmission times, the signal light will not be activated or activated too late. When the engine driver passes by a deactivated train signal light, the fallback procedure with reduced train speed and blown horn has to be executed.

Deployment Deploying hundreds or thousands of sensor nodes may be required by the application and can be very problematic and error prone.

In our use case less than twenty nodes are assumed, simplifying hardware deployment a lot. Software deployment during operation, like updates, is not permitted because the regular service must not be interfered.

Coordination To enable efficient communication between sensor and actor nodes, a good coordination mechanism is needed.

In our use case a automated WSAN is assumed, thus a modelling abstraction for decentralised coordination problems is required. To ease modelling of coordination problem a space-based approach shall be used because they supports spatiotemporal decoupling [20] and some space concepts also supports distributed systems where reliable communication cannot be assumed (e.g. LIME [57]), as for WSANs.

Mobility Especially in the robotics domain, mobile nodes are very common and have to be supported by WSAN protocols.

Since our use case is stationary, this thesis gives no special attention to mobility aspects.

2.2 Wireless Sensor Node Hardware

Typical WSAN hardware consists of a (possibly energy-optimised) MCU and a wireless radio. Wireless radios commonly operate in the Industrial, Scientific and Medical Band (ISM Band). The ISM Band specification defines radio frequencies which are free to use and the usage of those bands is regulated by the EN 300 220-1 [19], covering limitations on bandwith, centre frequencies, dissipated power and regulations for spectrum access techniques.

There are some COTS systems available which are intended for WSAN usage. The most well-established examples are:

MICAz The MICAz [51] node has been designed and manufactured by Crossbow Technology, Inc. (now MEMSIC Inc) and features an Atmel ATmega128L, a low-power 8 bit MCU, and a 2.4 GHz IEEE 802.15.4 compliant transceiver. From the software point of view, MoteWorks and TinyOS are supported.

TelosB TelosB [52] is a node from the same company as the MICAz and also has similar characteristics. Instead of the 8 bit MCU, a 16 bit Texas Instruments MSP430 is used as processor. For wireless communication also a 2.4 GHz IEEE 802.15.4 compliant wireless module is used and the TelosB supports TinyOS.

Waspmote The Waspmote [48] is designed and manufactured by Libelium Comunicaciones Distribuidas S.L. and is based on an Atmel ATmega1281, an 8 bit MCU. A wireless radio is not on-board, but a ZigBee socket is available to attach external wireless transceivers. An accelerometer, an external Real-Time Clock (RTC), a microSD card slot and a solar charger is provided by the node. As programming environment a modified version of the Arduino framework is used.

Seeeduino Stalker The Seeeduino Stalker [62], sold by Seeed Technology Ltd., offers, except of the accelerometer, similar features like the Waspmote, namely both are based on an Arduino-compatible Atmel AVR MCU, both provide a ZigBee socket for wireless communication, a solar charging circuit, an external RTC and a microSD card slot. One big advantage is that the hardware design is open source and has been published online at [62].

The suitability of existing hardware with respect to our use case will be analysed in Section 5.1.

2.3 Space-based Middleware

For the development of automated WSANs domain specific assumptions play an important role. Therefore the basic abstraction introduced by embedded OSs, like TinyOS, should be extended by mechanisms which ease modelling of coordination and offer decoupling in time, space and synchronisation. This is can be done by following a space-based computing approach and is explained in more detail in the following sections.

Basic terminology about decoupling, according to [20], is:

Space Decoupling Space decoupling denotes that interaction parties, in our use case the sensor/actor nodes, do not need to know each other

Time Decoupling Time decoupling means that it is not required that the interaction parties are active at the same time. This is especially helpful when connections cannot be assumed to be stable, like in WSANs.

Synchronisation Decoupling When publishing parties are not blocked during producing events, this is called “producer-side synchronisation decoupling”. Synchronisation decoupling denotes that also subscribing parties can be notified asynchronously, for example by a callback handler.

2.3.1 Linda Tuple Spaces

Linda [34] specifies one shared memory, called space, among all distributed parties. This shared memory holds arbitrary many tuples with typed values as values of these tuples. Tuples can be

retrieved from the space by template matching. Templates allow a selection of tuples with an equality comparison or typed wildcards. For example if there is a tuple $(\text{'temperature'}, 35.5, \text{'deg'})$ in the space, this tuple could be retrieved by a template like $(\text{'temperature'}, ?float, ?string)$.

The proposed operations on this space are "in", "inp", "rd", "rdp" and "out". In Linda, "in" denotes a blocking consuming read, "rd" a blocking non-consuming read and "out" a non-blocking write operation on the shared space. When issuing an "in" or "rd" with a template and no matching tuple is available in the space, the operation blocks the execution until a matching tuple can be read. Tuples in the space have no order and are therefore read indeterministically. The corresponding non-blocking operations of "in" and "rd" are "inp" and "rdp".

By using the space abstraction and the defined read and write operations, Linda tuple spaces offer space decoupling, time decoupling and producer-side synchronisation decoupling [20].

An illustrative example of a Linda shared space can be seen in Figure 2.2. Please note that each service may run on a different client side.

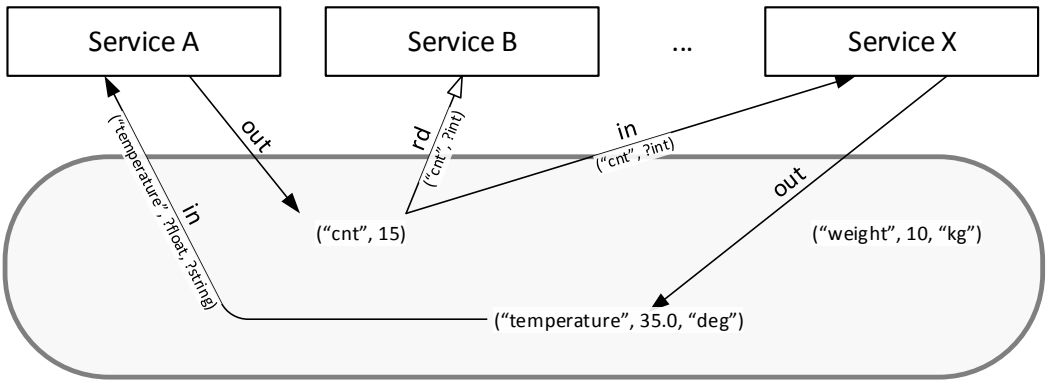


Figure 2.2: Linda [34] alike Tuple Space with Three Shared Tuples.

2.3.2 XVSM Spaces

XVSM [11, 44] (short for eXtensible Virtual Shared Memory) extends the Linda tuple space approach and proposes a peer-to-peer architecture. Data is stored in so-called "entries" and in contrast to Linda, where tuples are stored directly in the space, entries are stored in containers, allowing a logical grouping of data. These containers reside in the shared space and are managed by coordinators. Coordinators manage the access to containers. Typical coordinators are for example:

Random The "random" coordinator offers random access to the container.
FIFO A FIFO coordinator introduces a First In First Out (FIFO) structure on contained entries.
Linda The "Linda" coordinator offers Linda like template matching.

An example for an XVSM space one containers with a FIFO coordinator is shown in Figure 2.3.

Main operations on containers are: **take** for a blocking, consuming read, **read** for a blocking, non-consuming read and **write** for a non-blocking write operation. Additionally the concept of

aspects is introduced. Aspects are callback handlers which can be registered before and after operations are executed and are also able to modify passed data. With aspects, a notification functionality can be realised where a callback handler is called upon a certain event happens.

By the introduction of the notification mechanism, additionally to spatiotemporal decoupling, also synchronisation decoupling on the subscriber side is enabled.

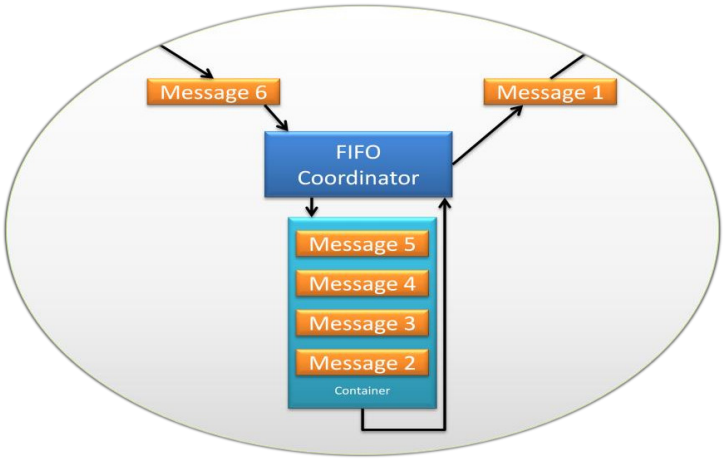


Figure 2.3: XVSM [11, 44] Space with a Container and a FIFO Coordinator. Image taken from [50].

The XVSM space abstraction serves as the basis for the proposed coordination abstraction, namely the Peer Model, described in more detail in Chapter 3.

2.4 Domain Specific Language (DSL)

A DSL [53] makes sense whenever problems shall be solved in a specific domain and the abstraction of domain-specific assumptions reduces the effort to solve these. This can either be done by embedding domain-specific assumptions into a General Purpose Language (GPL) or by inventing a new language from scratch.

In [53] the advantages and disadvantages of different DSL development approaches are summarised as follows:

Language Embedding

The domain specific specialisations are embedded into an existing host language.

Advantages

- Since existing languages are used, the development effort is modest.
- Often provides a powerful language because many features from the host language come for free.

- Development and debugging environments can be reused from the host language.
- Training costs for developers may be lower.

Disadvantages

- Because most host languages do not allow arbitrary syntactical extensions, the DSL syntax is far from optimal.
- Especially for overloaded operators, new semantics may be different and confusing when it is integrated in the host language because the embedded language may have different properties than the host language.
- Since DSL concepts are different from the host language's concept, error reporting is usually very poor.
- Domain-specific optimisations are hard to realise.

Language Invention

Language invention means that a new DSL is created from scratch.

Advantages

- Syntax can be close to the notation of the underlying model.
- Helpful error reporting can be implemented.
- Analysis, Verification, Optimisation, Parallelisation and Transformation (AVOPT) is enabled.

Disadvantages

- A high development effort is required.
- According to [53], DSL development often leads to incoherent designs.
- Typically language extensions are hard to realise because language processors are not designed to be extensible.

In this thesis, for the design of the DSL a language invention approach has been chosen to enable domain specific AVOPT and to bring the syntax close to the Peer Model specification shown in Chapter 3.

2.5 Related Work on Middleware Systems

Various middleware concepts for embedded systems and space-based middleware approaches have already been realised. Based on the requirements induced by the motivating use case, following selection criteria have been chosen:

Space-based The middleware should follow a space-based paradigm to ease modelling coordination of distributed systems, like for WSANs, and ensure decoupling in time and space to cope with unreliable connections.

Embedded An embedded implementation, running on MCUs, is mandatory. Systems only providing an implementation for mobile nodes like smartphones or Personal Digital Assistants (PDAs) are excluded because these implementations will not work for energy-aware embedded WSAN nodes.

WSN support Selected systems have to support a WSN (or WSAN) by using wireless radio communication over distances longer than 50 m. Sub-1GHz radios are preferred because of energy efficiency.

Energy Awareness The middleware has to support energy saving mechanisms to ensure a self-sustaining operation. In general, for systems following a Mobile Agent (MA) [64] paradigm this is not fulfilled since the transmission of the agent implementation implies a high network load, resulting in an energy consuming operation.

2.5.1 Overview

The selected related systems, which are covered in respective sections, are therefore:

- TCMote (Section 2.5.2)
- TeenyLIME (Section 2.5.3)
- PEIS (Section 2.5.4)

A graphical overview of the selection criteria and related work is shown in Figure 2.4.

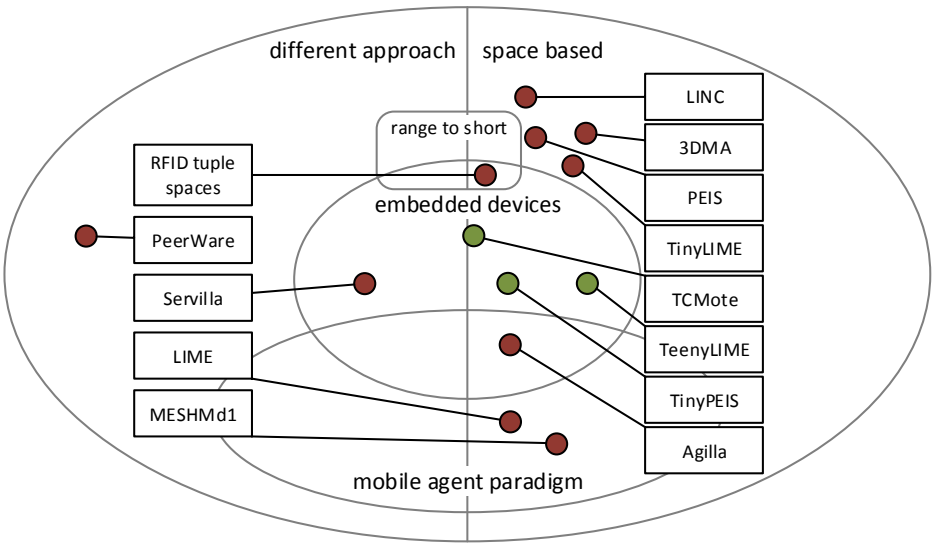


Figure 2.4: Related Work Overview. Systems marked with a green circle fulfil selection criterion and red marked systems are excluded.

Following related systems have been excluded because at least one of the requirements is not met:

3DMA 3DMA [27, 25, 24, 23, 26] is a space-based middleware supporting mobile computing and rule-based coordination. For 3DMA also a graphical modelling tool is available. However, 3DMA is only implemented in Java for mobile devices and thus, the "Embedded" requirement is violated.

Agilla Agilla [30, 29, 28, 31] is a Linda like space-based MA middleware. MAs are able to move and clone themselves and coordination is done over the tuple space. The MA architecture is implemented based on the transmission of application code, resulting in a high network load and thus, the "Energy Awareness" criterion is violated.

LINC LINC [56, 17] is a space-based middleware which uses a distributed set of bags to encapsulate tuples. To specify the behaviour of the system, production rules based on space operations are used. However, the prototypical implementation is Python-based and therefore violating the "Embedded" requirement.

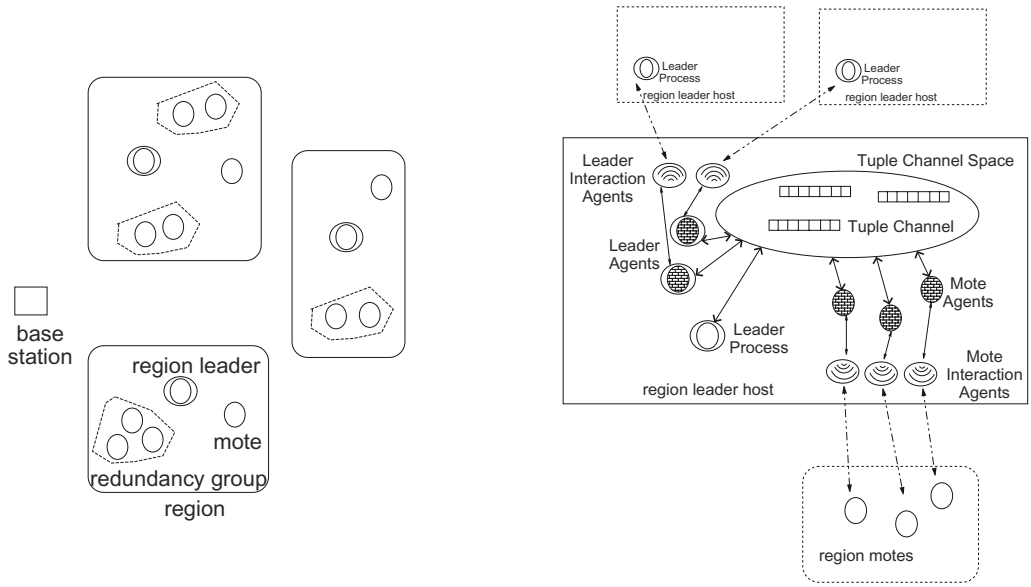
MESHMd1 MESHMd1 [37, 36] is a tuple space-based middlware following an MA approach and allows rule-based information diffusion. The prototype was realised in Java for mobile devices only and thus the system violates the "Embedded" requirement.

PeerWare PeerWare [35] follows an interesting approach using a distributed structured file-based database based on a Linda-tuple space in combination with a publish/subscribe mechanism. The implemented prototype does not run on embedded platforms which violates the "Embedded" criterion.

RFID Tuple Spaces RFID Tuple Spaces [49] follows a Linda like space approach where data is exchanged over rewritable memories of Radio-Frequency Identification (RFID) tags. Since communication is done over RFID the communication range strongly depends on the output power of used RFID readers and typically results in very short distances and thus, violating the "WSN support" criterion.

2.5.2 TCMote

In [16, 14] the architecture of TCMote, which is short for "Tuple Channel Mote", is introduced and in [15] it is extended by C-TCM, a DSL for TCMote. TCMote does not follow the classical Linda [34] tuple space approach but uses a so-called tuple channel space instead. This tuple channel space incorporates tuple channels as coordination and communication medium. A tuple channel offers an ordered many-to-many channel for tuple transmission. When requesting data from a channel, data is only consumed locally and the channel is not modified from other clients' point of view. Tuple channels are addressed by an attribute-based identification with a selection mechanism similar to Linda template matching but instead of tuples, the selection mechanism uses an attributed structure with equality comparison and wildcards. TCMote's network structure is based on a hierarchical approach where every sensor node belongs to a region with one leader. Those region leaders require a higher computational power to enable data aggregation.



(a) TCMote Operational Setting.

(b) TCMote Middleware Architecture.

Figure 2.5: TCMote Example Setting and Middleware Architecture. Images taken from [14].

Although the hierarchy is predetermined by region leaders, nodes are able to move to a different region, which may increase the performance of the network. An example of an operational setting is sketched in Figure 2.5a. The square called "base station" is a sink node which collects sensor data from the network. The rounded rectangles denote regions where nodes are grouped to and double circles mark region leaders and normal circles mark WSN nodes.

A schematic diagram of a region leader's communication is shown in Figure 2.5b. The region leader is denoted by the big rectangle and nodes in the same region are shown below in the dashed box. The region leader encapsulates the tuple channel space abstraction and virtual agents accessing the space. At the top of Figure 2.5b, communication with other region leaders is sketched.

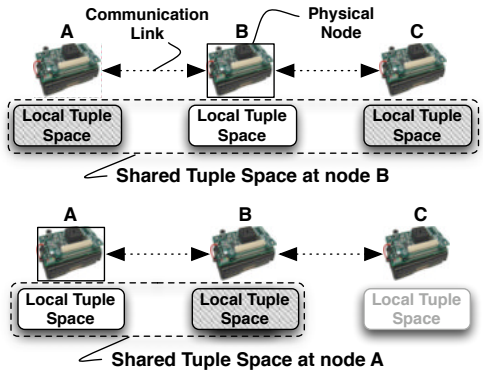
An extension of TCMote, called "TC-WSANs", is introduced in [4]. There the system is extended to additionally enable the integration of actors as super nodes like region leaders. This extends the TCMote approach by the support of semi-automated WSANs.

A major drawback of TCMote / TC-WSANs is that it is only able to operate in WSNs / semi-automated WSANs. As toolchain support an embedded C like domain specific language with code translation to C is provided.

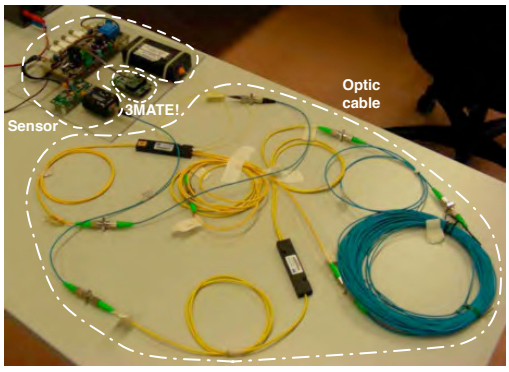
2.5.3 TeenyLIME

TeenyLIME [10, 55, 8] especially targets the operation of automated WSANs. It therefore adapts the LIME model [57], which uses a MA approach and binds a tuple space to each node and transiently shares them among all co-located nodes, to embedded platforms. Each node hosts

one local Linda [34] tuple space which is shared among its one-hop neighbours. A mechanism called "reactions" allows to register functions which are triggered whenever a tuple is inserted into the space that matches a specific template. The view of respective nodes with one-hop neighbours' shared tuple spaces is shown in Figure 2.6a.



(a) Shared TeenyLIME Spaces.



(b) TeenyLIME Node with a Fibre Optic Sensor.

Figure 2.6: TeenyLIME Space Concept and Hardware. Images taken from [9].

To enable time-sensitive applications with TeenyLIME, the time axis is divided into epochs and whenever a tuple is created, the current epoch is attached to the according tuple. A function then allows to get the number of elapsed epochs since the creation of a tuple.

TeenyLIME also adds a range matching functionality to the template matching. This mechanisms extends the classical Linda template matching, which only uses equality and wildcards, by adding comparisons like "$>$", "$\geq$", "$<$" and "$\leq$" to the template. This may reduce the network load for tuples which are only required when a certain threshold is exceeded since tuples are only transmitted upon need.

Special "capability tuples" allow to register handlers to templates. The registered handlers are executed whenever a "in" or "rd" with the same template as registered is requested. This allows to execute actions upon template requests and to respond with a tuple of the requested form. This greatly improves the nodes' energy consumption for scenarios where certain measurements are only needed upon request.

An example for this would be a temperature sensing node. Without the "capability tuples" feature, the sensor node has to sample the temperature all the time and create a tuple for each measurement to ensure data freshness. By using "capability tuples", the temperature readout function is registered for the template requesting the temperature and thus, temperature readout is only performed upon need, namely when a temperature value is requested.

TeenyLIME is implemented by following an event-based approach on top of TinyOS. In [9] the development of a node specialised for monitoring heritage buildings is described. One node with a fibre optic sensor attached is shown in Figure 2.6b.

Summing up, TeenyLIME offers an adapted Linda tuple space implementation with some interesting extensions for embedded applications. However, it lacks in toolchain support as TeenyLIME only offers a library for TinyOS.

2.5.4 PEIS

The PEIS [60, 59, 6, 7], short for “Physically Embedded Intelligent Systems”, middleware targets the integration of different sensor, actor and sensor/actor nodes into one “PEIS ecology”. The PEIS middleware deals especially with three issues: heterogeneity, self-configuration and knowledge sharing. A schematic of an example for a PEIS ecology, with different interacting participants ranging from powerful desktop devices to embedded MCUs, can be seen in Figure 2.7a and a picture of PEIS devices in the real world is shown in Figure 2.7b.

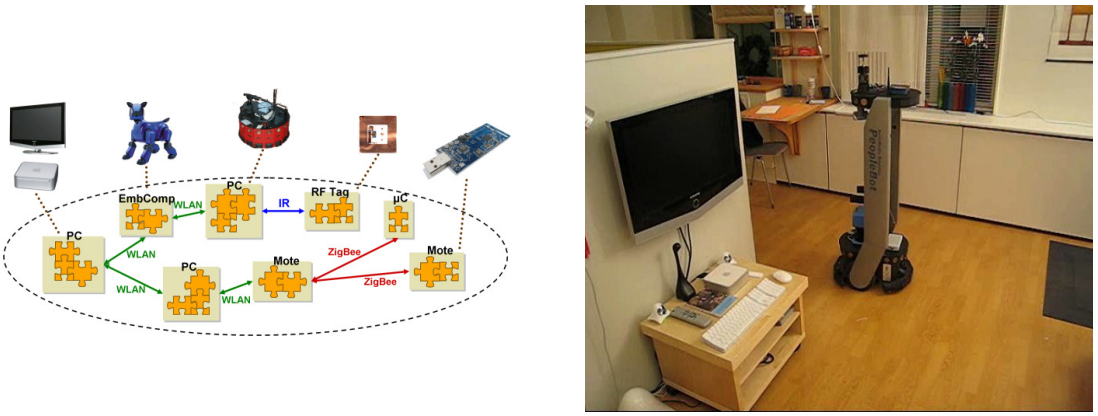
To cope with heterogeneity there are different implementations for computationally stronger platforms, like PCs or PDAs, and for smaller platforms, like MCUs. Since the focus of this work lies on WSANs, the implementation for smaller platforms, called TinyPEIS [5], builds the central part for this analysis.

TinyPEIS [5] is implemented on top of TinyOS and provides a space abstraction with key/-value pairs as tuple elements. The keys have to be predefined for the whole ecology in an ontology file. In contrast to the regular PEIS implementation, those keys are represented as integer values (instead of strings) and the maximum allowed data size of tuples is reduced to fit radio packets below 100 bytes. To react to specific tuples in the space, keys can be subscribed. Whenever a tuple with that key is inserted into the space the subscriber is notified.

To integrate TinyPEIS into the regular PEIS ecology a special connector component called “tinybridge” is used. This connector builds the bridge between embedded devices and stronger devices like PCs and PDAs. Additionally, this component has to translate between string-keyed tuples from the regular implementation and integer-keyed tuples from the embedded implementation.

The TinyPEIS kernel reference implementation has a memory footprint of about 35 kB Read Only Memory (ROM) and 2.5 kB Random Access Memory (RAM).

The PEIS middleware is a tuple space-based middleware for heterogeneous networks, however it lacks in toolchain support since TinyPEIS is only provided as a library on top of TinyOS.



(a) Schematic of a PEIS Ecology.

(b) PEIS Embedded in the Real World.

Figure 2.7: PEIS Example Ecology. Images taken from [60].

2.5.5 Summary

All of the selected systems offer a space (or at least space like) abstraction. TeenyLIME extends the adapted Linda model (LIME) by energy saving features like “capability tuples” and “range matching”. TCMote also offers a DSL, embedded in C, and a code generation. Although, either the related systems do not offer a toolchain reaching from design to implementation for embedded devices or they are not suited for automated WSANs networks.

A summary of the compared systems with respect to the following evaluation criteria is shown in Table 2.1.

Feature	Systems		
	TCMote	TeenyLIME	PEIS
Space-based coordination	✓	✓	✓
Rule-based coordination	×	×	×
DSL	✓	×	×
Code Generation	✓	×	×
Graphical Notation	×	×	×
Low-Power Mechanisms	?	✓	?
Semi-automated Networks	✓	✓	✓
Automated WSAN	×	✓	?
Network Topology	Hierarchical	Peer-to-Peer	Peer-to-Peer
Supported Platforms	TinyOS	TinyOS	TinyOS

Table 2.1: Feature Matrix of Compared Systems.

Rule-based Coordination Rule-based coordination means that coordination can be purely realised by implementing rules based on space operations. For example, Linda template matching or TeenyLIME range matching does not count as rule-based coordination because only reaction to the input are possible and the output has to be implemented in application code.

DSL This feature describes if the system offers an either textual or graphical DSL.

Code Generation This states if a code generation is provided. This feature implies that there is a DSL as a basis. Code generation eases the development process because modelled software can directly be translated to executable code and mistakes coming from translation by hand are avoided.

Graphical Notation Describes if a graphical representation is provided. Graphical notations support developers in software reviewing and debugging.

Low-Power Mechanisms States if the implementation supports energy saving mechanisms like sleep mode usage. This is a very important feature for embedded WSAN nodes because these are typically battery operated. The “?” denotes that it is not explicitly mentioned whether energy saving is supported or not.

Automated WSAN Here it is stated if the system also works for automated WSANs. The middlewares where it is not explicitly stated if they support automated WSANs or not are marked with a "?".

The support of automated WSANs is required by our use case.

Network Topology The network topology describes for which kinds of networks the system is applicable. For WSNs a star topology is very common but mesh networks are on the rise as well. Some systems just provide an one-hop (neighbourhood only) network, which is referred as peer-to-peer network, and more complex networks can be implemented in software, nevertheless this results in a higher software complexity at the application layer.

Supported Platforms This feature enumerates the supported platforms. Here, "TinyOS" refers to nodes with TinyOS support, like the MICAz, TelosB, etc. Please note that only embedded platforms are listed here. All of the considered systems also offer a bridging to a PC or PDA implementation.

CHAPTER 3

The Embedded Peer Model (ePM)

As WSANs typically consist of many nodes which sense and react to sensed values, these systems are highly concurrent. Communication and synchronisation of those nodes and their applications is known as coordination. Solving such coordination problems can be a tedious task and to tackle these problems, the Peer Model [46, 45, 42] comes in handy because it is designed to ease modelling and implementation of coordination in concurrent and distributed systems.

The Peer Model works are as follows: Information is encapsulated in entries which are stored in containers. Wirings consist of a guard, service and action section. The guard contains a set of guard links which request entries of a specified type, which must fulfil a query, from an assigned container. When all guard links of a wiring are fulfilled, specified services are executed and action links deliver produced entries to containers afterwards. A peer has two containers, namely a Peer-In-Container (PIC) and a Peer-Out-Container (POC) representing the input and output stages of the peer, and a set of wirings which implement the peer's behaviour.

The ePM is derived from the Peer Model, adding some assumptions and restrictions to ensure an efficient realisation for embedded devices. However, the used specification is widely equivalent to the Peer Model specification described in [46, 45]. The detailed ePM specification is covered in more detail in the following sections.

Since MCUs are usually based on a Harvard architecture, which splits memory into data (RAM) and program memory (ROM), software modification during run-time is either not possible or not recommended. Because of that dynamic behaviour like adding entry types, wirings to peers or deploying peers at run-time is explicitly not supported in the ePM.

3.1 Entry

A Peer Model entry represents stored data which includes a type, coordination properties and optional application data. The coordination properties section is split in framework and application coordination properties. As the name proposes, coordination properties are used for coordination logic and can be seen as a label/value pair.

In the ePM, for application coordination properties only Integer (a 32 bit integral value) and Boolean (tt or ff) values are allowed. Because of limited data storage size of MCUs the application data section is prohibited.

To avoid entry duplication, entries are assumed to be immutable, which means that whenever an entry has to be modified (which typically occurs rarely compared to read operations) the entry has to be copied, modified and emitted again.

3.2 Coordination Specific Datatypes

To introduce a notion of time, enable communication with other peers and the correlation of distributed tasks, some basic datatypes have to be introduced:

Timestamp

An ePM timestamp describes an absolute point in time and can also be not set (`nil`). The concrete timestamp format, range and granularity has to be defined by the according implementation.

Address

Since one of the key features is modelling of distributed applications, addressing mechanisms are required. For the embedded case the address is either a node name or `ANY`, resulting in a broadcast transmission.

Flow

To make parallel executed tasks easier to handle, execution is grouped into so-called flows. The mechanism which groups tasks into such flows is called flow correlation and wirings only access entries with a compatible flow, namely the same flow or where the flow is not set.

3.3 Coordination Properties

The following coordination properties are specified by the ePM framework and attached to each entry by default:

Time To Start (TTS)

The TTS property specifies when an entry gets active and is of type “timestamp”. When an entry with a TTS property set to a future point in time is created, it is only valid after the TTS is reached. The default value for the TTS is `nil` which means that the entry is valid at any point in time.

Time To Live (TTL)

Analogue to the TTS, the TTL specifies how long an entry is valid. The datatype is also timestamp and the default value is `nil`. Here the `nil` value indicates that the validity of the entry does not expire. Entries with an exceeded TTL are invalid and cannot be accessed anymore. The concrete behaviour which has to be executed when an entry has exceeded has to be defined by the framework. For the ePM a simple clean-up behaviour which simply deletes expired entries is designated.

FLOW

The FLOW property enables the flow correlation mechanism for entries and is therefore of type "flow" (as specified in Section 3.2). The default value for created entries is the current flow which is determined by the flow used for the guard links.

Destination (DEST)

To realise distributed applications, entries can be exchanged between peers by setting the DEST property. The DEST property is of type "address". When the DEST property is not set, the entry transportation is not triggered and the entry resides in the container it was written to.

3.4 Query

A query consists of a mandatory entry type, a selection and a count specification. At first, queries require that entries match the specified entry type. If the entry type is different to the specified type, the query is not fulfilled.

Then an optional selection is applied which is explained in Section 3.5. After the successful application of the selection, entries are chosen based on a specified count operation. If the count is not specified, it defaults to "exactly one". As count operation either a comparison like lower-equals ("$\leq$"), greater-equals ("$\geq$"), equals-to ("$=$"), in-range ("$\in [lower, upper]$"), `all` or `none` is possible. For lower-equals, greater-equals and equals-to, a positive integer has to be given and for the in-range two positive integers have to be specified as limits. The in-range operation also includes the interval's bounds. The `all` count literally equals to "greater equal zero" (which is not allowed for the greater-equals operator since zero is not a positive integer). The `none` count specification works a bit different to other operators since it is only satisfied if no entry fulfilling given type and selection is available in the assigned container.

3.5 Selection

Selections allow to choose entries by defining conditions which have to be satisfied by the entries' application coordination properties. For the ePM a simple per-entry condition checking is designated which checks simple expressions on entries. Allowed expressions are comparisons

of integer values (using $<$, $>$, $\leq$, $\geq$ or $=$) and boolean operators (“and”, “or” and “not”) resulting in a boolean value. For every entry the expression is checked and if it is satisfied, the entry is added to the resulting set.

3.6 Container

Containers keep entries and provide read and take operations to select contained entries. The container concept is based on space-based middleware containers, like XVSM [11, 44], and therefore, supported container operations are **read** and **take** with a query (as specified in Section 3.4). A **read** operation just reads selected entries while the **take** operation reads selected entries in a consuming way, which means that they are removed from the container.

For the ePM the container also handles the TTS, TTL and DEST properties. When an entry’s TTL expires, the entry is simply cleaned up and no further actions, like exception handling, are triggered in the embedded case.

3.7 Entry Collection

For the ePM, the entry collection is a container which does not provide framework coordination properties (like the TTS, TTL and DEST). Entry collections are used as local storage for wirings (described in Section 3.10).

3.8 Link

A link connects two containers and comprises a space operation (**read** or **take**) and a query.

Links are used for guards, actions, service inputs and service outputs. Guard links connect a container (PIC or POC) with an entry collection, action links connect an entry collection with a container, service input links connect a wiring’s entry collection to a service input and service output links connect the service output to a wiring’s entry collection.

Graphical Representation

(a) Take Operation. (b) Read Operation.

Figure 3.1: Examples for Peer Model Links.

Figure 3.1a shows two examples: a **take** operation which reads exactly one entry of type “entry_1” satisfying the condition “entry_1.weight $\geq$ 5”. In Figure 3.1b a **read** operation is depicted which ensures that no entry of type “entry_2” is available.

3.9 Service

Services operate on an entry collection with service input and output links and encapsulate coordination and application logic. A service is allowed to access entries by **read** or **take** operations (as service input). Then, based on the input, services may create, modify and emit new entries which are then moved to the entry collection (service output). To glue coordination logic and application logic together, services can also call code outside of the ePM. For service inputs and output links, the ePM only allows queries selecting a type and using an `all` count.

3.10 Wiring

Wirings are the control logic of the Peer Model. Each wiring operates on an entry collection and has a guard, a set of services and an action. At first, guard links are checked and then, if all are satisfied, the wiring fires, which means that all services and action links are executed.

Since embedded devices typically do not support multi-threading, at most one wiring is active at any time and wirings are executed as one atomic block.

Guard

A guard comprises multiple guard links and a guard link consists of a space operation (**read** or **take**), a query and an assigned container. If the container is not explicitly set, it defaults to the PIC. Optionally FLOW correlation and TTS checking can be suppressed. Guards are checked sequentially in the specified order and the first fulfilled guard which has a not-`nil` FLOW sets the current flow for the subsequent wiring execution.

To allow the creation of entries upon peer start up, a special guard condition called “initial” is introduced. A wiring containing an “initial” guard link is only executed at the beginning of the execution.

When all guards are fulfilled, they are executed and put read or taken entries into the wiring’s entry collection.

Service Invocation

After successfully executing guards, service invocation is started. All services are executed sequentially in the specified order. At first service input links are processed, the the service implementation is executed and finally service outputs move created entries to the entry collection.

Action

At the end of the wiring execution, when all services were invoked successfully, the ePM distributes all entries with a set DEST property. Then action links are handled by delivering specified entries from the entry collection to assigned containers. If no explicit container assignment is given, it defaults to the POC. The ePM only allows queries with a type and an `all` count for action links and all entries which reside in the entry collection after wiring execution are cleaned up automatically.

Graphical Representation

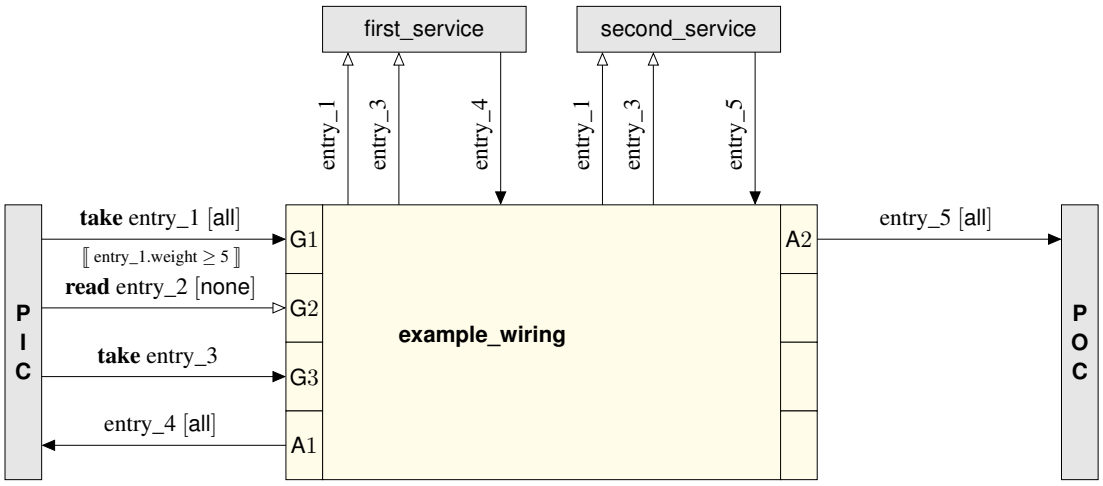


Figure 3.2: An Example Wiring.

The wiring shown in Figure 3.2 takes all entries of type "entry_1" which satisfy the condition "entry_1.weight $\geq$ 5", checks that there is no entry of type "entry_2" and takes exactly one "entry_3" entry from the PIC. When this is successful, the services "first_service" and "second_service" are executed sequentially and finally action A1 delivers all "entry_4" entries to the PIC and A2 delivers all "entry_5" entries to the POC. Links with a G_n or A_n in the slot box denote guard or action links and the number denotes the sequence of execution per category.

3.11 Peer

A peer comprises two containers, namely one PIC and one POC, and a set of wirings. The PIC represents the peer's input stage of and the POC its output stage. Typically wirings' guards are linked to the PIC and actions to the PIC or the POC.

Graphical Representation

The example peer shown in Figure 3.3 instantiates two wirings, namely the wiring shown in Figure 3.2 and a second wiring which takes one Entry of type "entry_4", executes a service called "transformation_service" and writes one Entry of type "entry_6" to the POC.

3.12 Topology

The ePM also introduces a topology specification which pins peers down to real processors, in our case MCUs. One processor can run at most one peer.

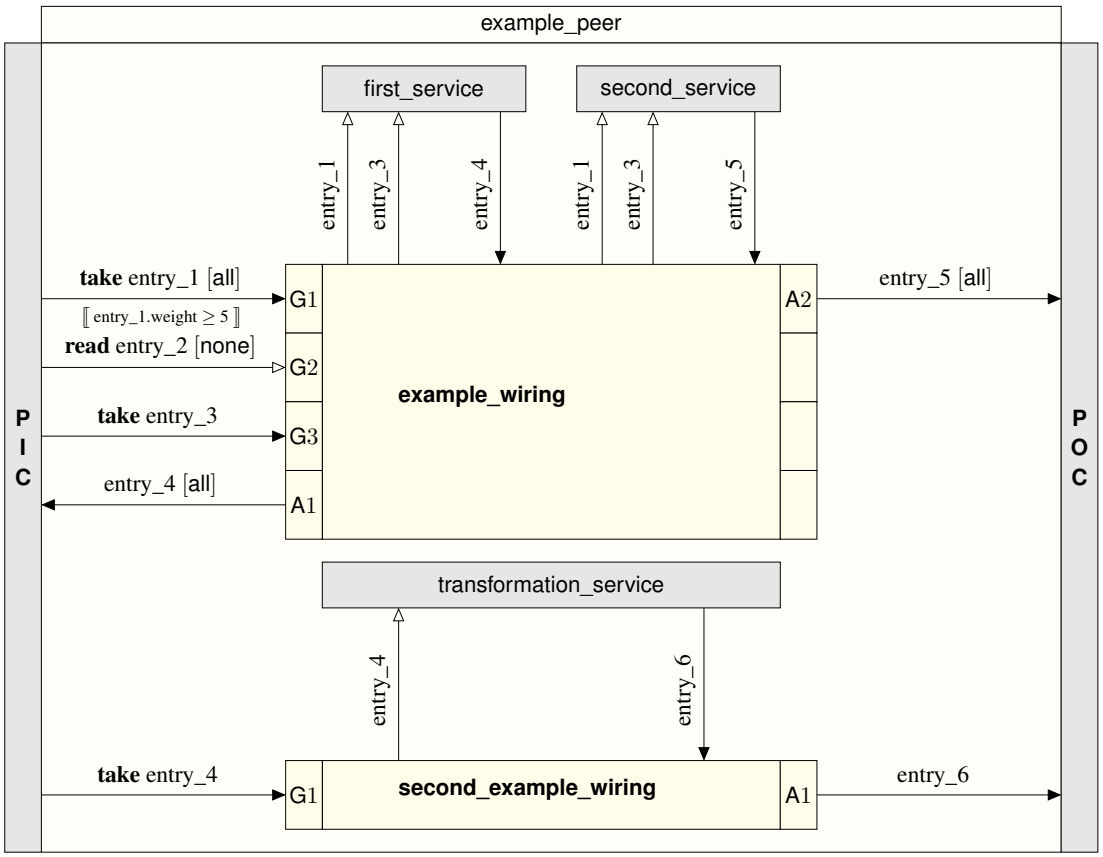


Figure 3.3: An Example Peer.

CHAPTER 4

Domain Specific Language Specification

In this chapter the development of the DSL for the ePM is covered. In the future this DSL may be extended to an enterprise version of the Peer Model as well. Therefore the requirements on the DSL are extensibility (allowing additional syntactical elements to be added by framework extensions), compatibility with the graphical DSL and an easily readable, understandable and concise syntax.

For the syntactical specification the Extended Backus-Naur Form (EBNF)[1] according to [39] is used.

4.1 Design Decisions

To fulfil the requirements stated above, the following design decisions have been made:

Extensibility To reach language extensibility there exists a mechanism in the used parser framework [2] to conditionally include or ignore syntactical elements based on a configuration file. For the purpose of WSANs, only a restricted language set tailored to the needs of the ePM is described here. Further lingual features for more advanced versions can be added upon need.

Readability To improve the readability of the DSL the syntax has been designed in the style of ADA [40], which is known for good readability [66]. In contrast to other common languages like C or Java, ADA [40] uses a more verbose syntax and special attention is given to structural analogies.

[1] Please note that » `{ A }` – « is equivalent to » `A , { A }` «.

[2] "pegthon" taken from `https://github.com/anwesender/pegthon/`

Concise The DSL is focused on coordination tasks only. Non-coordination related services have to be integrated using service wrappers and are called by the `call`-statement in the service implementation.

Executability In order to leave room for interpretation, the execution semantics is not specified in detail for the DSL. This enables frameworks to slightly change the behaviour in some points.

Language Type For the ePM a new DSL is designed from scratch to smoothen the way for AVOPT in future.

4.2 General Syntactical Elements

Common used EBNF meta-identifiers are:

```
upper letter    = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J'
                | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T'
                | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' ;
lower letter    = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
                | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'
                | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' ;
pos digit       = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
digit           = '0' | pos digit ;
alpha           = upper letter | lower letter ;
alnum           = alpha | digit ;
identifier      = ( alpha | '_' ) , { ( alnum | '_' ) } ;
positive int    = pos digit , { digit } ;
integer         = [ '+' | '-' ] , { digit }- ;
boolean         = 'true' | 'false' ;
address         = 'processor' , '(' , identifier , ')' ;
primitive type  = 'int' | 'bool' | 'flow' | 'time' | 'address' ;
primitive value = integer | boolean | address ;
operation input = 'read' | 'take' ;
epm selection   = (* specified in "4.6 Selection" *) ;
selection         = '[[' , epm selection , ']]' ;
entry count     = '[' , ( ( '>=' , positive int )              (* at least *)
                        | ( '<=' , positive int )               (* at most *)
                        | ( positive int , '..' , positive int )  (* range *)
                        | 'all'                                    (* >= 0 *)
                        | 'none'                                (* none test *)
                        ) , ']' ;
query           = identifier , [ selection ] , [ entry count ] ;
```

4.3 Entry

For an ePM entry the specification of arbitrary many coordination properties is possible. One property in the entry declaration consists of a label (name), a primitive datatype (Integer or

Boolean) and a default value. When no coordination data is required, the `empty` keyword has to be used.

The DSL of an entry syntax in EBNF is as follows:

```
entry codata    = identifier , ':' , primitive type , ':=' , primitive value ;
entry           = 'entry' , identifier , 'is' ,
                    'coordination' , 'data' , 'is' ,
                    ( 'empty' | { entry codata , ';' }- ) ,
                  'end' , 'entry' , identifier , ';' ;
```

Two examples for entries declared in DSL are:

1. An entry of type “trigger” without any coordination data:

```
entry trigger is
    coordination data is empty
end entry trigger;
```

2. A “sensor” entry holding a train axle counter with a default value of zero:

```
entry sensor is
    coordination data is
        axles : int := 0;
end entry sensor;
```

4.4 Service Implementation

As for the ePM DSL, a declarative service specification has been designed. It allows the creation of new entries, based on incoming entries, creation of new flows and calling external ANSI C methods (these have to be registered in a configuration file before). Examples for the service implementation are included in the service wrapper examples below in Section 4.5.

```
assignment      = identifier , '=>' , expression ;
emod emit       = 'emit' , [ '(' , [ 'count' , '=>', positive int ] , ')' ] ;
emod copy       = 'copy' , '(' , 'from' , '=>' , identifier , ')' ;
emod set        = 'set' , '(' , assignment , { ',' , assignment } , ')' ;
entry modifier  = '.' , ( emod emit | emod copy | emod set ) ,
                        [ entry modifier ] ;
emit entry      = 'create' , '(' , identifier , ')' , [ entry modifier ] ;
external call   = 'call' , identifier , [ '(' , assignment , ')' ] ;
new flow        = 'new' , 'flow' ;
service stmt    = ( emit entry | external call | new flow ) , ';' ;
service impl    = { service stmt } ;
```

4.5 Service Wrapper

The ePM service requires an input and output specification and a service implementation. The input and output specification has to be defined explicitly to enable easy service composition and black-boxed service encapsulation.

```
service input   = operation input , query , ';' ;
service output  = 'emit' , query , ';' ;
service impl    = (* specified in "4.4 Service Implementation" *) ;
service         = 'service' , identifier , 'is' ,
                    [ service input ] , [ service output ] ,
                  'begin' ,
                    service impl ,
                  'end' , 'service' , identifier , ';' ;
```

Two examples for service wrappers are:

1. The "`blink_led`" service implements a simple Light Emitting Diode (LED) blinking behaviour. Upon execution, the first actuator (in our case a LED) is activated and deactivated after five seconds. Please note that the led blinking behaviour only works with an according wiring which delivers the emitted "`digital_out`" to the POC.

```
service blink_led is
  emit digital_out[all];
begin
  -- service implementation
  create(digital_out)
    .set(actuator => 1, state => true)
    .emit
    .set(actuator => 1, state => false, tts => now +5s)
    .emit;
  -- end of service implementation
end service blink_leds;
```

2. The following service takes a digital input (entry "`digital_in`"), emits an entry which activates the ignorance of further digital inputs for the next 30 secs and emits a "`sensor_transport`" entry with a new flow.

```
service sensor is
    take digital_in[all];
    emit sensor_transport[all];
    emit digital_in_ignore[all];
begin
    create(digital_in_ignore)
        -- that's the time how long 'digital_in's are ignored
        .set(ttl => now +30s)
        .emit;
    create(sensor_transport)
```

```
        .set(flow => new flow)
        .emit;
end service sensor;
```

4.6 Selection

For the ePM a limited selection syntax, tailored to the needs for embedded systems, is used (as described in Section 3.5). The limited selection only allows basic conditions on the entry's coordination data. Examples for queries are included in the wiring examples below in Section 4.7.

Comparisons allowed in a selection are simple integer expressions with plus, minus, multiply and divide operations, boolean expressions with "and", "or" and not operations, integer and boolean comparisons and entry members as operands.

```
entry acc       = identifier , [ '[' , positive int , ']' ] ;
entry memacc    = entry acc , '.' , identifier ;
int par         = '(' , int addsub, ')' ;
int lvlbase     = integer
                | entry memacc
                | int par ;
int lvl0neg     = '-' , int lvlbase ;
int lvl0        = int lvl0neg
                | int lvlbase ;
int muldiv      = int lvl0 , { ( '*' | '/' ) , int lvl0 } ;
int addsub      = int muldiv , { ( '+' | '-' ) , int muldiv } ;
int comp        = int addsub , int cmp , int addsub ;
comparison      = int comp
                | bool comp ;
bool par        = '(' , bool or , ')' ;
bool lvlbase    = comparison
                | boolean
                | entry memacc
                | bool par ;
bool lvl0neg    = '!' , bool lvlbase ;
bool lvl0       = bool lvl0neg
                | bool lvlbase;
bool and        = bool lvl0, { 'and', bool lvl0 } ;
bool or         = bool and , { 'or' , bool and } ;
bool comp       = bool or , bool cmp , bool or ;
epm selection   = bool comp ;
```

4.7 Wiring

An ePM wiring is specified as a template which can be instantiated in the peer implementation afterwards. The wiring specification comprises the guard, the service section and the action.

```
guard entry     = operation input , query,
                  { 'suppress', ('tts' | 'flow') } ,
                  [ 'from' , identifier ] ;
guard special   = 'initial' ;
guard link      = guard entry | guard special ;
service         = 'running' , 'service' , identifier ;
action link     = 'deliver' , query , [ 'to' , identifier ] ;
wiring          = 'wiring' , identifier , 'is' ,
                  'begin' , { guard link , ';' } , { service , ';' } ,
                  { action link , ';' } ,
                  'end' , 'wiring' , identifier , ';' ;
```

Following example wiring "send_event" first asserts that there is no "ack" and at least one "event" entry in the PIC. If this is the case, the "event" entry is removed from PIC and written to the entry collection and the "send_event" service is executed. Finally all "event" entries emitted by the service are delivered to the PIC and all "digital_out" entries are moved to the POC.

```
wiring send_event is
begin
    read ack[none];
    take event;
    running service send_event;
    deliver event[all] to PIC;
    deliver digital_out[all];
end wiring send_event;
```

4.8 Peer

An ePM peer may instantiate arbitrary many wirings. For the instantiation of the wirings only the wiring's name needs to be specified.

```
peer            = 'peer' , identifier , 'is' ,
                  'begin' ,
                  { 'instantiate' , 'wiring' , identifier } ,
                  'end' , 'peer' , identifier , ';' ;
```

This code defines a peer with two wirings, namely 'init_forwarder' and 'forward_event'.

```
peer forwarder is
begin
    instantiate wiring init_forwarder;
    instantiate wiring forward_event;
end peer forwarder;
```

4.9 Topology

The topology maps the peers to real WSAN nodes, called processors. For every WSAN node one line including its name and the name of the peer have to be specified.

```
peer            = 'topology' , identifier , 'is' ,
                  'begin' ,
                  { 'processor' , identifier , 'runs' ,
                    'peer' , identifier , ';' } ,
                  'end' , 'topology' , identifier , ';' ;
```

1. The following topology specifies one processor running the "`main`" peer.

```
topology example_single is
begin
    processor MAIN runs peer main;
end topology example_single;
```

2. A topology consisting of four processors running three different peers is shown in the following:

```
topology example is
begin
    processor LN_1_SENSOR   runs peer sensor;
    processor LN_2_FW       runs peer forwarder;
    processor LN_3_FW       runs peer forwarder;
    processor LN_4_ACTUATOR runs peer actuator;
end topology example;
```

CHAPTER 5

Hardware

To bridge the gap between the theoretical model and real world application, a WSAN node hardware is required to run the reference implementation shown in Chapter 6. A first analysis of existing hardware has been done in Section 2.2 and in this Chapter a hardware selection for a prototype implementation meeting the requirements induced by the motivating use case is carried out.

5.1 Requirements

Since the WSAN nodes must work self-sustaining in the field they must be powered by rechargeable batteries, and include some sort of energy harvesting circuit. They have to be optimised regarding energy consumption. The wireless modules must be exchangeable to enable benchmarking of different wireless transceivers and for logging purposes a memory card slot should be available as well.

It is recommended that the board operates with a voltage of 3.3 V since most wireless chips require that voltage and single cell Lithium Polymer (LiPo) batteries have a voltage of 3.7 - 4.2 V, which comes handy because no step-up conversion to e.g., 5 V is required, increasing efficiency and reducing components. A hardware encryption is also desired to enable the implementation of basic security concepts for wireless communication. Additionally, a RTC is recommended for a precise time base and a source of wake-up interrupts.

Mandatory

- Powered by rechargeable batteries.
- Energy harvesting circuit, e.g. solar powered battery charger.
- Memory card support for logging purposes.

- Low energy consumption.
- A wireless radio socket for exchangeable wireless radio modules.

Desirable

- Board voltage of 3.3 V.
- Hardware support for encryption e.g., Advanced Encryption Standard (AES).
- RTC support.

5.2 Overview of Related Hardware

In Section 2.2 well-established WSAN nodes are mentioned. For the prototype implementation the MICAz and the TelosB are excluded because they incorporate a 2.4 GHz wireless module which contradicts the low-energy consumption; and the exchangeability of wireless radios requirements is not provided.

The Waspmote and the Seeeduino Stalker are similar and both fulfil the requirements. For the first prototype, the Seeeduino Stalker has been selected because its hardware designs are open-source, thus enabling a better debugging.

Finally an own hardware has been developed to further decrease the node's energy consumption and enable encryption by choosing a MCU with AES acceleration.

5.3 Seeeduino Stalker

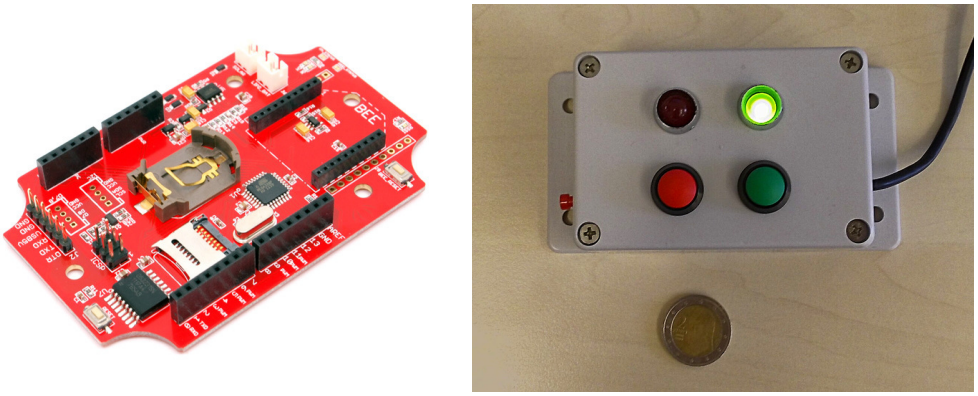
Board

Since most wireless radios currently work with a voltage of 3.3 V an Arduino-compatible hardware which operates with 3.3 V has been selected for the evaluation. The Seeeduino Stalker v2.3, an Arduino Pro compatible controller which can be seen in Figure 5.1a, seemed to be a good choice for this purpose because it fulfils all mandatory and desired requirements stated in Section 5.1.

Wireless Radio

For the sake of simplicity, at first a ZigBee PRO compatible [2] module has been chosen for wireless communication. ZigBee PRO has the advantage that it works straight out of the box and can easily be configured by a modified Hayes command set[41].

Although ZigBee allows easy configuration, a central coordinator is mandatory and multi-hop routing requires so called router nodes. Both, router and coordinator nodes consume a lot more energy than end nodes and messages exchanged between end devices have to pass the central coordinator which makes ZigBee only feasible for star networks and in general not for linear node arrangements as requested in our use case.



(a) Seeeduino Stalker v2.3. Taken from [62]. (b) Boxed System under Evaluation.

Figure 5.1: Arduino Compatible Target.

5.4 LOPONODE

In contrast to available COTS boards the energy consumption can be optimised by changing the MCU and wireless radio to energy-optimised versions or removing unnecessary Digital Input/Output (Digital I/O) like power LEDs. For this purpose a custom Printed Circuit Board (PCB) has been designed specifically for the LOPONODE project to fulfil its low-power requirements. The assembled PCB can be seen in Figure 5.2a and a boxed node is shown in Figure 5.2b.

MCU

A comprehensive list comparing MCUs regarding energy consumption and Peer Model Framework requirements can be found in [38]. As a result of this comparison the Energy Micro Giant Gecko "EFM32GG" has been chosen and it is located in the middle of the board (the black square-shaped Integrated Circuit (IC)), shown in Figure 5.2a.

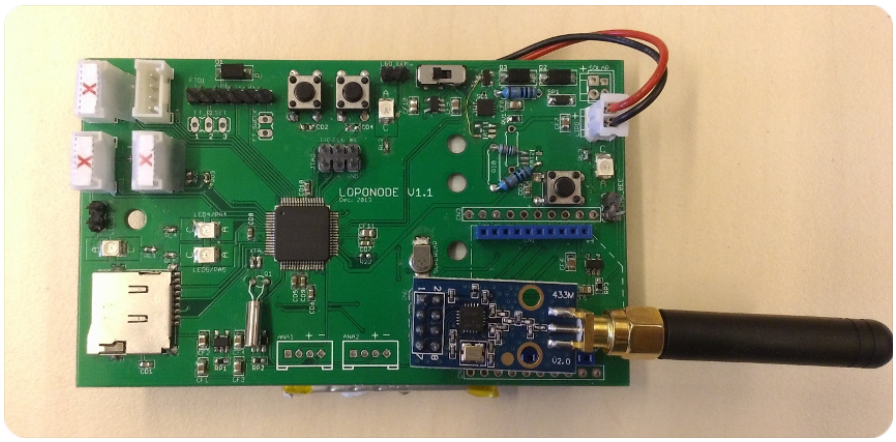
Energy Harvesting

For energy harvesting reasons a solar charging circuit has been designed on top of the CN3063 charging IC and is laid out for the usage of LiPo batteries. The charging current has been adjusted to 380 mA which equals to about 0.2 C[1] for a 2000 mAh battery.

LiPo batteries have been chosen because they are easily available and have a rather high energy density and no memory effect.

The solar charging circuit takes place on the top right of the LOPONODE board, shown in Figure 5.2a. On the back side of the board a LiPo battery is attached which is connected by the cable with the black and red wires.

[1] The unit C represents a charging current of 1 A per Ah capacity.



(a) LOPONODE with CC1101 radio.

(b) Fieldtest-ready LOPONODE.

Figure 5.2: LOPONODE Platform.

Wireless Radio

As wireless radio a CC1101 transceiver from Texas Instruments has been selected since it is widely available and supports required ISM Band frequencies. This wireless module can be seen in Figure 5.2a on the bottom right (the blue PCB with an external antenna attached). Since bare-metal access to the International Organization for Standardization (ISO) Open Systems Interconnection model (OSI) Physical Layer (Layer 1) is possible, the design of customised ISO OSI Data Link Layer (Layer 2) implementations is enabled.

For backward compatibility also a ZigBee radio socket has been added to the PCB. The ZigBee socket is below the CC1101 wireless module in Figure 5.2a.

Other Peripherals

For debugging purposes two LEDs have been added to the board. To support external peripherals, like the wheel sensor for the railway use case, external LEDs, other sensors for WSNs like temperature or accelerometers, sockets with the so-called “Grove” receptacle (on the upper left in Figure 5.2a) have been added to the board. A microSD card socket has also been incorporated to support logging of the application’s behaviour.

CHAPTER 6

Reference Implementation

To enable source code generation for various platforms, documentation generation and interfaces to formal verification and deployment tools, a compiler which transforms ePM DSL to arbitrary output has been implemented.

6.1 Peer Model Compiler

Architecture

An overview of the compiler can be seen in Figure 6.1. The frontend, which contains the Parser, reads the supplied DSL files and generates an Abstract Syntax Tree (AST) as intermediate structure. Then the middleend checks and modifies the given AST. Finally the backend, which comprises the code generation part, generates source code, depending on the selected target, out of the modified AST. A configuration file allows to configure language features for the frontend, set options for middleend modules and selects and configures the backend target.

The whole compiler reference implementation is realised in Python 3 [63]. Python 3 has been chosen because it is a well-established (usually interpreted) programming language available for most desktop platforms with a broad library support. For the parser frontend the "pegthon" framework is used[1].

Configuration

The configuration file uses INI syntax [2] and allows to configure parts of the framework, parser language features, execution of middleend modules and backend selection.

An example configuration can be seen in Listing 1.

[1] Taken from `https://github.com/anwesender/pegthon/`.

[2] `https://technet.microsoft.com/en-us/library/cc731332.aspx`; Accessed: 2015-04-01

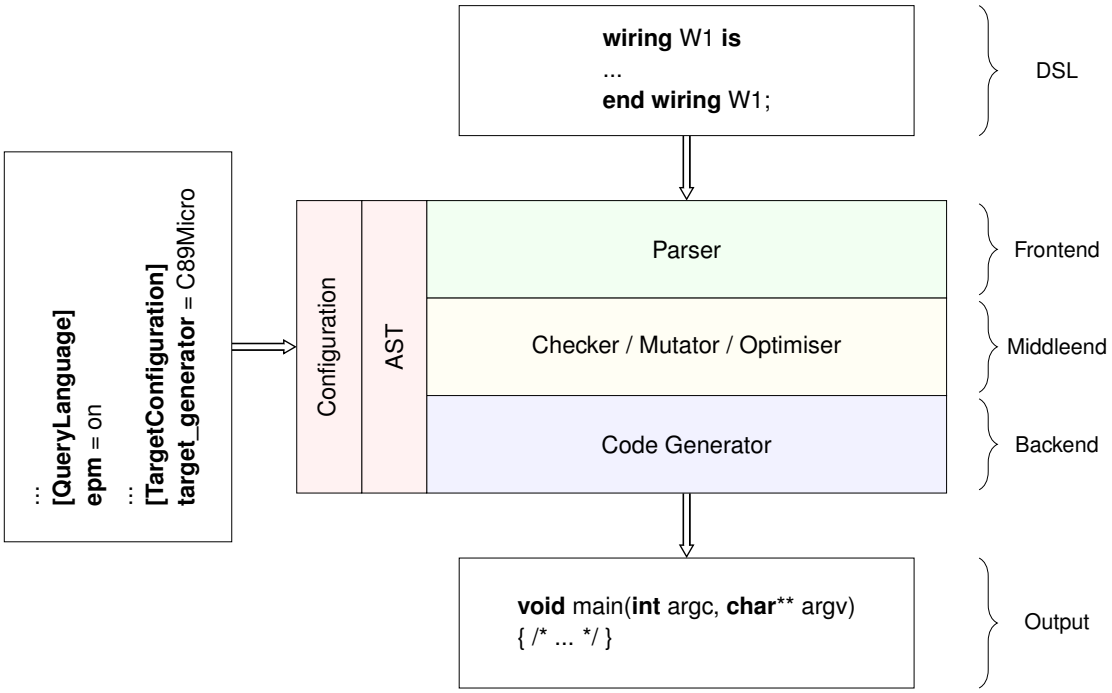


Figure 6.1: Compiler Structure.

AST

The AST is automatically generated based on the attributed[3] grammar and input by the parser framework.

Frontend

The compiler frontend includes the modular parser which is configured by a given configuration file. Depending on which language features are set the parser is adapted to. Because of the modular parser, the query language and the service language are exchangeable and the backend has to support selected language features.

Lexical analysis is done during parsing using a regular expression approach. For parsing a Parsing Expression Grammar (PEG) Parser written in Python is used. PEGs have several advantages [33] compared to Context-Free Grammars (CFGs) since they are unambiguous, faster parsable[4] and do not require preprocessing steps.

[3] Note that attributed isn't meant in the classical sense of an attributed grammar. Here attributed means that every parsed element which should be contained in the resulting AST gets a 'name' attribute resulting in an AST member after parsing is completed.

[4] When using algorithms like `packrat`[32].

```
[Folders]
project = .
output = ./output-c89m-lnv1
framework = pmc3-frameworks/c89micro/build/lnv1-efm32gg232

[FramworkCodata]
tts = on
; ...

[QueryLanguage]
epm = on

[MiddleEnd]
entry_name_checker = on
service_name_checker = on
; ...

[TargetConfiguration]
target_generator = C89Micro
```

Listing 1: Example Configuration File for the Peer Model Compiler Reference Implementation.

Middleend

The middleend does checking and optimisation tasks which are divided into platform-dependent and platform-independent parts. Platform-independent checking tasks could be checking if used entries are fully specified in the entry file or guard conditions are unsatisfiable. Platform-dependent checking is used for checking compatibility with e.g. embedded versions which maybe won't support subsets of the query semantics or similar. Platform-independent optimisers are used for general optimisation like constant folding, reducing entry collections to used entries only. Platform-dependent optimisers can optimise run-time/framework specific things like sequential execution optimisation.

Backend

In the compiler backend the whole code generation is done. Code generator modules get the processed AST and configuration as input. Not only executable code could be generated, also design documentation, formal verification or visualisation outputs are possible.

To reduce the effort of backend programming for textual output, e.g. C, C++, Java or LaTeX code, a template engine is highly recommended.

6.2 ANSI C Embedded Implementation

The ePM ANSI C implementation is intended for the use with Reduced Instruction Set Computer (RISC) MCUs with little memory and computational power. Little memory and computational power means a RAM size of about 1 - 16 kB, a ROM size of about 16 - 128 kB, a typical clock

frequency of about 16 - 48 MHz and a throughput of about one Dhrystone Million Instructions Per Second (DMIPS) per MHz[65].

6.2.1 Overview

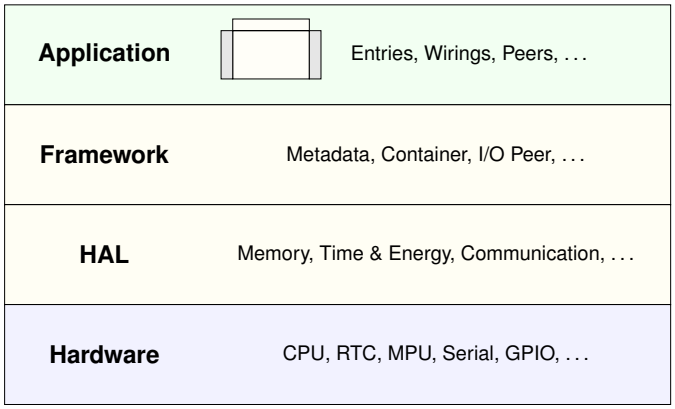


Figure 6.2: ANSI C Framework Structure

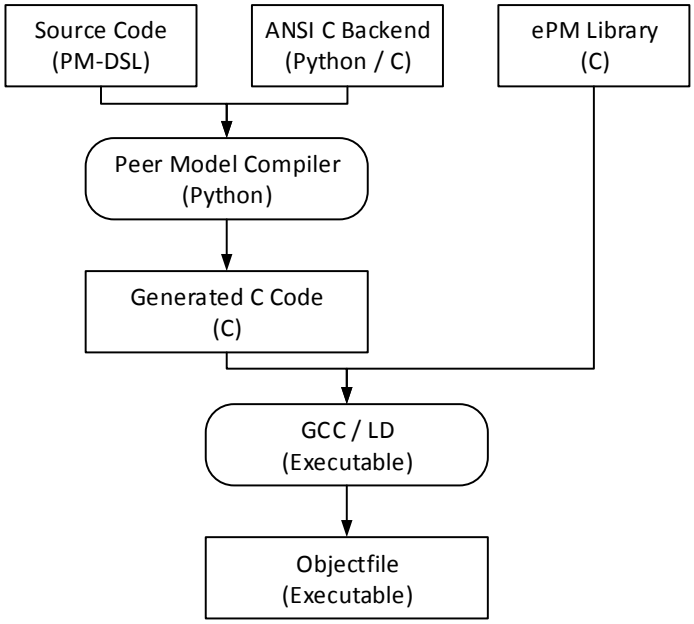


Figure 6.3: Compilation Process for the ANSI C Implementation.

As can be seen in Figure 6.2, the ePM implementation is split into an application part, which is generated from an ePM DSL, a framework part which offers ePM functionalities like entry metadata, Digital I/O and transport peer functionality, memory pools and a HAL instrumenting the hardware.

To generate an executable file out of an ePM DSL, the process displayed in Figure 6.3 has to be followed: At first the application has to be realised in the DSL and a configuration file has to be created to select the ANSI C output. Then the Peer Model Compiler is invoked, generating the ANSI C code and a "Makefile" with according paths and configurations. To generate an executable it's recommended to use the generated "Makefile". Otherwise the C compiler toolchain has to be used to generate object files out of the ePM framework and the generated code. After linking all object files together to one binary, this can be flashed onto the MCU.

6.2.2 Embedded Peer Model Framework

The framework description covers required memory pool implementations, ePM specific datatypes (like the basic entry type, flow identifier, entry collection and basic wiring and peer types), entry serialisation and the ePM container implementation.

Memory Pool

The typical MCU targeted by the ePM framework does not have a Memory Management Unit (MMU) on-board. This implies that any dynamic memory management has to be handled by the framework. Although, in case the platform has a, preferably hardware-accelerated, dynamic memory allocation (`malloc` and `free`) these functions are used instead of the (usually less efficient) memory pools.

The ePM framework implementation offers two kinds of memory pools:

Reference Counted Memory Pool The reference counted memory pool uses fixed memory block sizes and adds one byte to the end of every allocated memory cell to count the current number of usages of this memory block. This kind of memory alignment is sketched in Figure 6.4a. The provided methods by the memory pool are:

get Allocates a new memory block with a reference counter of zero.

use Increases the reference counter by one.

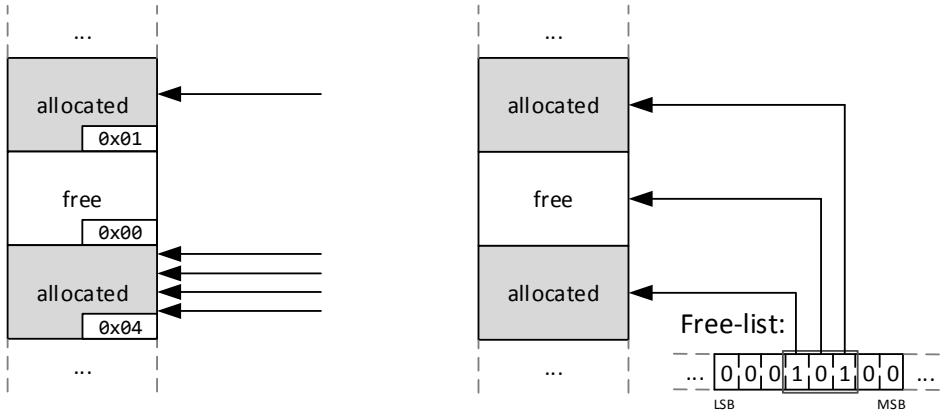
release Decreases the reference counter by one. In the case that the reference counter goes below zero, the memory block is freed.

To ensure a correct behaviour, before using an allocated memory cell the "use"-method has to be executed. Then the memory block can be read or written and after the end of the usage, "release" has to be called.

The reference counted memory is currently only used for the entry storage. Entries are assumed immutable so they can be referenced by more than one container or entry collection at

the same time. This greatly improves the performance of **read** operations and other cases where entries are used more than once, because a copy-on-read procedure is not necessary.

Simple Memory Pool For memory cells where the lifetime can easily be determined, e.g. for linked list pointers, a simpler memory pool implementation is used. The simple memory pool also uses static memory block sizes and administrates the blocks by maintaining an used/free list. This used/free list is implemented with a bit-field where each bit corresponds to one memory cell. With this approach even double freeing can be detected. A schematic of this can be seen Figure 6.4b. Please note that a sequential allocation of two memory blocks does in general not imply that their memory is arranged consecutively.



(a) Reference Counted Memory Pool. (b) Simple Memory Pool.

Figure 6.4: Memory Pool Types.

Peer Address

Each peer is addressed by a processor address and a peer identifier. This is sufficient for an unique identification because in the ePM each executed peer is unique per processor and processors are unique regarding their address.

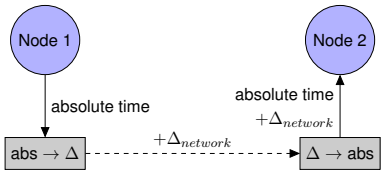
Flow Identifier

For the ePM, the flow identifier is a pair of a flow counter and the origin's peer address. This combination gives a quasi-unique[5] identifier. Depending on the estimated lifetime of a flow, the bit-width for the flow counter has to be chosen accordingly. For the use cases described in Chapter 7 a flow counter bit-width of 16 bit has been considered as enough.

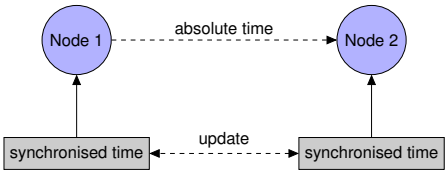
[5] Here, "quasi-unique" means that it is unique over a limited amount of time because of possible overflows.

Time Abstraction

The time abstraction uses the representation from the HAL, specified in Section 6.2.3. and requires functions for getting the current time, doing an `is_before` comparison and request a wake-up interrupt at a given point of time. Before entering a sleep mode, a wake-up interrupt is registered for the earliest TTS or TTL instant.



(a) Timestamp Offsetting.



(b) Time Synchronisation.

Whenever entries with a set TTS or TTL property are sent over the network, these properties necessitate special care. If no time synchronisation is implemented at the target platform, TTS and TTL values have to be converted to relative values upon sending and converted back to absolute values upon receiving, as visualised in Figure 6.5a. To ensure a correct behaviour, the network latency has to be considered for the usage of distributed TTS and TTL values.

In case that a synchronised time is available, it is recommended to directly transmit absolute timestamps based on the synchronised time, as shown in Figure 6.5b.

Entry Base

The basis for an ePM entry are an entry type identifier, a DEST of type 'Peer Address', a FLOW of type 'Flow Identifier', a TTS and a TTL of type 'Timestamp', which are called framework coordination properties.

Entry Type Identifier The entry Type Identifier is realised by an unsigned integer of the required size. That means, if there are less than 256 entry types used, a `uint8_t` will be sufficient.

Concrete Entry Types Concrete entry types (C structs) are generated by the ePM compiler backend and are inherited in an object-based development way. That means that every entry `struct` contains at least the coordination properties section from the "entry base" struct defined by the framework.

Further members are then appended, according to the application coordination properties section defined in the DSL. Since application coordination properties types are restricted to the primitives Integer and Boolean, the size of an Entry struct can be determined at compile time [6]. Since entries are allocated using a memory pool with a static block size, the block size is the maximum size of all concrete entry types.

[6] In contrast to string members which may have a dynamic length.

Serialisation Entries are serialised to a binary stream. Basically members are serialised in the same order as they are saved in their according `struct` but data structure alignment (which modern compilers usually activate as an optimisation) is eliminated. This can be seen in Figure 6.6.

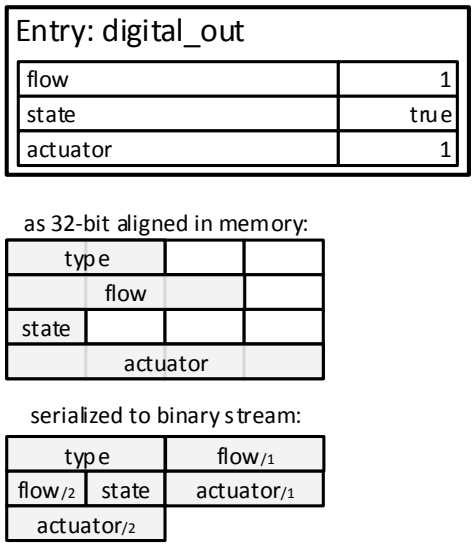


Figure 6.6: Entry Serialisation.

Container

Containers act like a tuple-space container and provide the storage for entries and query dispatching mechanisms. For the ePM reference implementation, the container governs over the lifetime of entries, which means that memory allocation, reference counting and de-allocation is handled by the container which owns the according entry. Therefore containers are implemented as a linked list in combination with a reference counting memory pool for entries and a simple memory pool for linked list elements.

To reduce MCU usage, the containers maintain an entry-query record per existing flow. The entry-query records contains the total number of entries, in the container, fulfilling a guard query for every guard query used in peers' wirings. These entry-query records are updated on every write and take access and on TTS and TTL interrupts. This mechanism is illustrated by executing "`wiring_x`", which is listed in Listing 2, and the entry-query matching is visualised in Figure 6.7. In this example two entries ("`state`" and "`digital_in`") are stored in the PIC. Both entries fulfil one query requested by the guard of "`wiring_x`" and thus, "`wiring_x`" is dispatched.

Entry Collection

For the entry collection a simple linked list has been used. Since the lifetime of linked list elements can easily be determined, the simple memory pool implementation is used as memory

```
service service_x is
    take digital_in[all];
    emit digital_out[all];
begin
    create(digital_out)
        .set(state => true, actuator => digital_in.sensor)
        .emit;
end service service_x;

wiring wiring_x is
begin
    take digital_in [[ digital_in.state == true ]];
    read state      [[ state.value == true ]];
    running service service_x;
    deliver digital_out[all];
end wiring wiring_x;
```

Listing 2: ePM Example Code for Illustrating Wiring Execution.

allocation.

Wiring

A wiring encapsulates one entry collection and processing functions for the wiring logic. The processing functions cover the guard processing routine, executing container take/read operations, the service calls and action distribution routine.

When the guard block is fulfilled, as it happened in Figure 6.7, according entries are moved from the container to the wiring's entry collection, like in Figure 6.8. Then the services are executed sequentially, Figure 6.9, and finally, the action block distributes entries from the entry collection back to containers which is shown in Figure 6.10.

Peer

An ePM peer owns two containers, the PIC and the POC. Both containers are allocated in the peer initialisation phase. Additionally a peer dispatching routing is provided which checks which wirings are fulfilled and have to be dispatched. This is done by matching the entry-query records to the wirings' guard blocks.

The ePM implementation uses tick-less scheduling and a peer's execution cycle, which is triggered upon MCU wake-ups arising from RTC, Digital I/O or transport interrupts, looks like the following and is illustrated in Figure 6.11:

1. Dispatching of the HAL, e.g. fetching inputs and messages from the transport abstraction.
2. Cleanup of entries where the TTL has been expired.
3. Iteration over the entry-query records, matching to wirings' queries and execution of the wiring, as shown in Figure 6.7, Figure 6.8, Figure 6.9 and Figure 6.10.

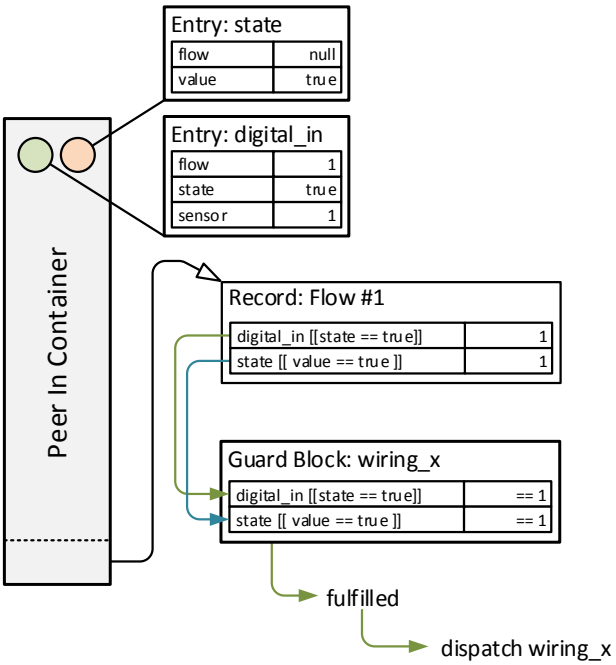


Figure 6.7: Entry-Query Record Mechanism.

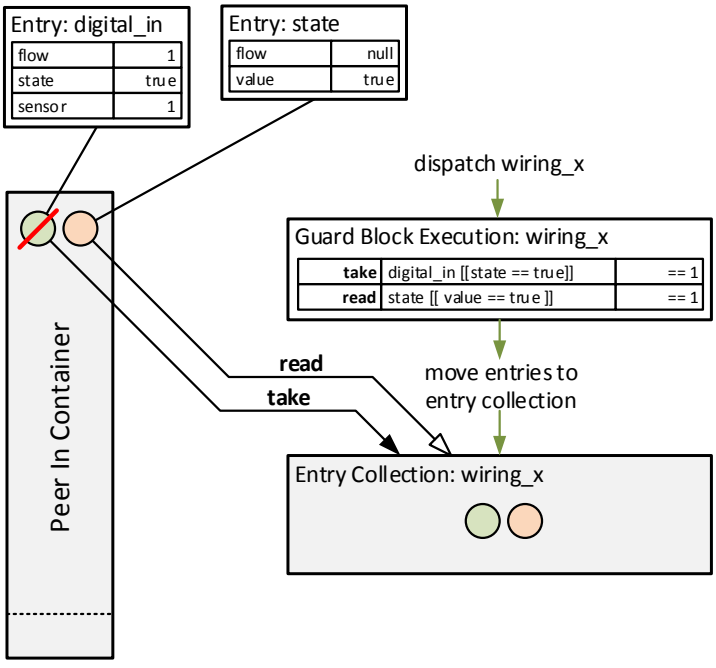


Figure 6.8: Wiring Execution: Guard Block.

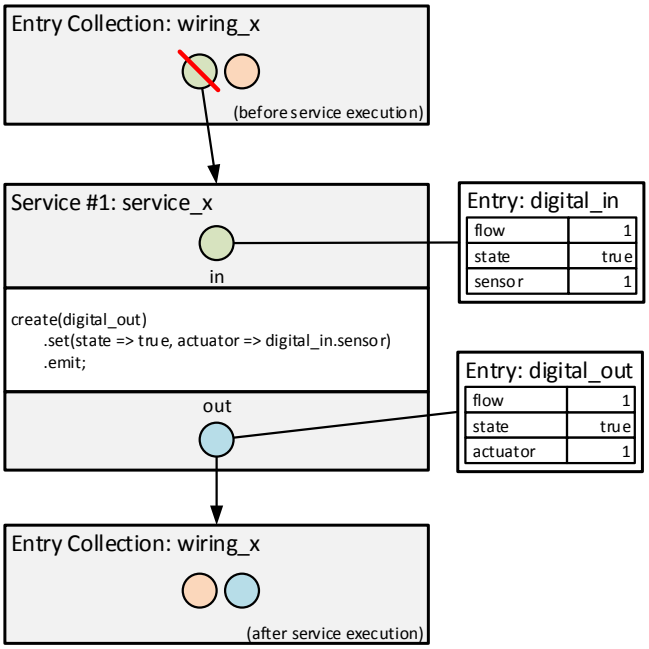


Figure 6.9: Wiring Execution: Service Invocation.

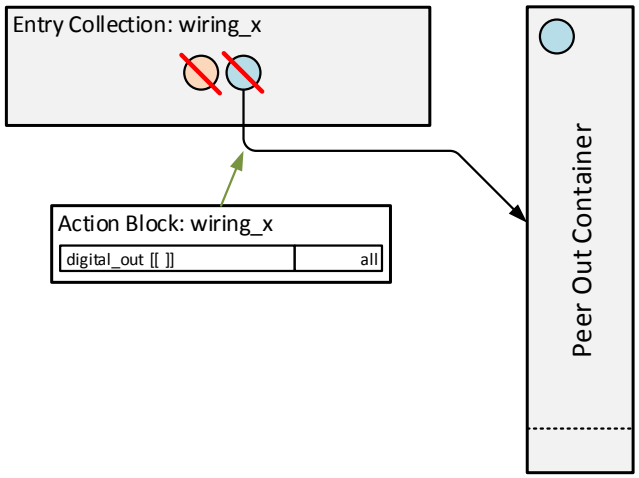


Figure 6.10: Wiring Execution: Action Block.

4. Again, dispatching of the HAL, e.g. setting outputs and delivering messages to the transport abstraction.

5. Set the next wake-up time to the minimum of TTL and TTS values of all entries in the PIC and POC.

6. Enter an appropriate sleep mode.

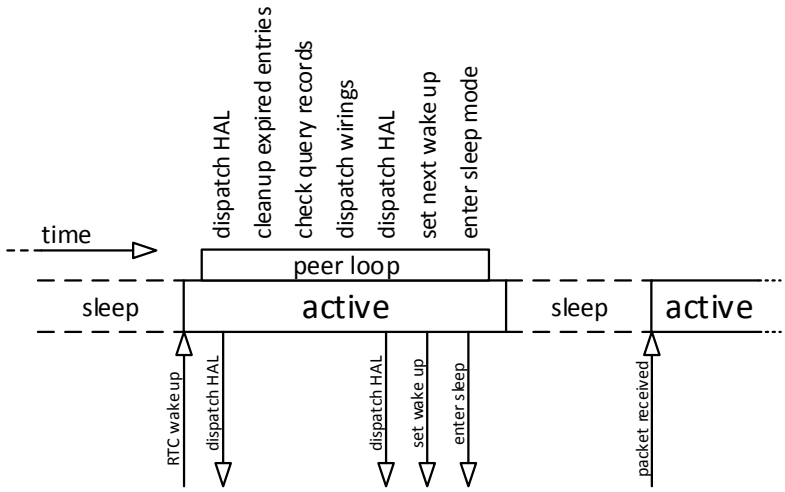


Figure 6.11: A Typical Peer Execution Cycle.

6.2.3 Hardware Abstraction Layer

The HAL is grouped into following functional units:

Memory Management

If the MCU comprises no MMU, at least an one-way memory allocation has to be implemented for the used platform. Then included memory pools are used for handling dynamic entry, record and container allocation. For the one-way memory allocation only a method with the requested size as parameter, which returns a pointer to an allocated block of that size, has to be implemented. In case the platform has a MMU and supports dynamic memory allocation, the provided methods are used and nothing else has to be implemented for memory management.

Time

This part of the HAL offers a representation of time and a “wake up at” mechanism. Required clock granularity and bit-width of timestamps strongly depends on the application and hardware restrictions. Additionally to the type of timestamps, `timestamp_t`, the time abstraction has to provide following methods:

`get_current_time(void)`
Returns the current time as `timestamp_t`.

`advance_time_by(timestamp_t time, seconds_t s, milliseconds_t ms)`
Returns the given time increased by `s` seconds and `ms` milliseconds.

`is_before(timestamp_t first, timestamp_t second)`
Returns `true` if the second timestamp is past the first one.

`set_next_wakeup(timestamp_t wake_up)`
Sets the next point in time when the MCU should wake up. After this call, entering a sleep mode is reasonable.

It is highly encouraged to use a RTC as a basis for this abstraction, as these typically support wake-up interrupts and have a rather low clock drift.

Energy Management

Modern MCUs typically support an energy management mechanism where the CPU enters a different state where the execution of code is stopped and clocks of different hardware modules are deactivated to save energy. Which hardware modules are deactivated strongly depends on the used hardware and sleep mode. For the hardware abstraction layer a sleep mode where the RAM content is still retained has to be chosen, but depending on the application and HAL implementation, Digital I/O and timers could be deactivated if they are not required. For this part of the HAL only a `enter_sleep_mode(void)` method has to be implemented.

Please note that there are also platforms, Complex Instruction Set Computer (CISC) architectures in general, which do not have sleep mode functionalities, but reduce the core frequency to save energy. On such platforms the framework will run as well but the energy saving will not be as good as on platforms with sleep mode support.

Transport Abstraction

The required methods for the transport abstraction are:

`initialize(processor_address_t own_address)`
Initialize the wireless radio with given address. If the wireless radio does not support addressing in hardware, the address should be stored to discard incoming messages which are for other nodes.

`send(entry_t data)`
Send given entry, the destination has to be taken from the DEST property.

`register_receive_callback(void (*handler)(entry_t data)))`
Register a callback that is called whenever an entry is received. Data received by the callback is then fed into the PIC.

The internals of the transport routines are in general either based on an asynchronous serial interface, called Universal Asynchronous Receiver Transmitter (UART), or on a synchronous serial interface, namely Serial Peripherial Interface (SPI). Since both have a slightly different mode of operation, these are covered in a slightly higher detail here:

UART based wireless radios Wireless radios with asynchronous serial communication normally have fully implemented network stack which covers ISO OSI Transport Layer (Layer 4), ISO OSI Network Layer (Layer 3), ISO OSI Layer 2 and ISO OSI Layer 1. This means that addressing, routing, Logical Link Control (LLC) and Media Access Control (MAC) is completely handled by the wireless module. An illustration of the connection to the wireless module is given in Figure 6.12a. To notify the MCU about received packages, a UART receive interrupt should be used. The best example of such radio modules are IEEE 802.15.4/ZigBee transceivers.

SPI based wireless radios Radios which require synchronous communication mostly only cover ISO OSI Layer 1 and some functionalities to ease an implementation of ISO OSI Layer 2. The connection to the wireless module is sketched in Figure 6.12b. In contrast to the UART based version, SPI communication requires an additional wake-up line to signal the MCU that a wireless packet has been received. This wake-up line has to be connected to an external interrupt pin at the MCUto ensure proper operation due to sleep modes. Examples of such wireless radios are the "Texas Instruments CC1101", the "Nordic Semiconductor nRF905" or the "Silabs Si4432". Even though all of them are able to operate on the same radio frequencies, every chip uses a slightly different communication interface and implement different wireless functionalities.

Please note that SPI based wireless radios typically offer a better performance because SPI communication can achieve higher transmission frequencies[7] and MCUs usually have a few special pins which allow a wake-up from all sleep modes whereas the UART receive interrupt in general only allows a wake-up of "IDLE"-like sleep modes.

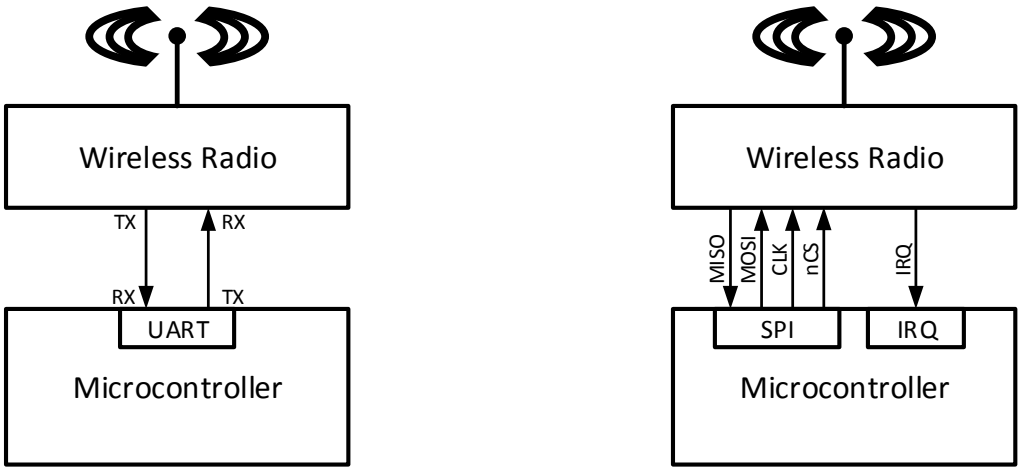
6.2.4 Supported Platforms

Seeeduino Stalker / Atmel AVR 328P

A reference implementation of the ANSI C framework has been realised for the Atmel AVR 328P without the usage of the Arduino library. Since this MCU builds the core of the very popular Arduino UNO, Mini, Nano and some other boards, this implementation of the ePM framework can be used for those as well.

Memory Management Like most MCUs the Atmel AVR 328P does not have a memory management or memory protection unit. Therefore the memory management is done by using a self written one-way memory allocation and memory pools.

[7] This is because there is a dedicated clock line and therefore no oversampling has to be done.



(a) UART-based Wireless Module.

(b) SPI-based Wireless Radio.

Figure 6.12: Typical Wireless Radio Configurations.

Time Since the Seeeduino Stalker's on-board RTC, a Maxim DS3231, only allows seconds as the finest granularity for wake-up commands and regular Arduino boards do not provide a built-in RTC, a Timer-based implementation has been chosen for time functionalities.

Energy Management For energy saving the "IDLE" mode, a sleep mode where timers and Digital I/O are still active, is entered whenever possible. A deeper sleep mode could not be chosen because then the timer-based wake-up implementation would not work and Digital I/O would be disabled as well.

The energy consumption could be further improved when Digital I/O is not needed for the use-case and `Timer2` is used in asynchronous mode as wake-up source. Then the "Power-save" or "Extended Standby" sleep modes could be used.

Transport To communicate with other nodes a UART transport interface has been implemented for the ZigBee platform and for the WizziMote implementation specified in [22].

LOPONODE / EnergyMicro EFM32

For the LOPONODE hardware, as described in Section 5.4, the HAL has been adapted accordingly and other parts of the ANSI C ePM framework are the same as in the Seeeduino Stalker implementation.

Memory Management Like the Seeeduino Stalker, the EFM32GG does not offer a MMU. Therefore a platform-specific one-way memory allocation has been implemented and the memory pools from the ePM framework are used for dynamic memory management.

In contrast to the Atmel AVR MCU, the EFM32GG offers a Memory Protection Unit (MPU) which allows to register up to eight memory regions which may be protected. This functionality has not been used for the ePM implementation but this could be added in future to protect memory regions against modification from misbehaving user services.

Time The EFM32GG MCU offers an internal "Real Time Counter" and a "Backup Real Time Counter". For the time hardware abstraction the "Backup Real Time Counter" has been chosen because it offers higher bit width for the counter value, namely 32 bit. Therefore a 32 bit unsigned integer is used to represent timestamps and the chosen counter granularity is about 4 ms (to be precise: a 256 Hz counter is used). These chosen timestamp and counter settings uniquely cover a time-span of about 194 days.

Energy Management To save energy the Energy Micro processors offer four sleep modes. Depending on the required peripherals an appropriate sleep mode has to be chosen. For the reference implementation the lightest sleep mode has been selected to maintain debug logging support. When debug logging and other peripherals are not actively used, a deeper sleep mode could be used as long as the RAM content is still retained.

Transport For the LOPONODE a rudimentary LLC and MAC layer has been implemented for the SPI-based CC1101 wireless transceiver. Because of the simple implementation of the lower layers, the performance is greatly improved and more control over the radio can be given to upper layers.

6.2.5 Middleend

In the Peer Model Compiler ANSI C middleend simple checking like the existence of used entries, entry members, services, wirings, peers and processors and simple feasibility checks for service-wiring interrelations, for example when a service assumes existence of entries in the entry collection which are not captured by any guards, is done. Furthermore the ANSI C implementation uses an access optimisation which links entries required by the service from the entry collection to an array where they can be directly accessed by the service implementation.

6.2.6 Backend

The backend is implemented using the `texthon` templating engine[8]. At first required entries are generated. An example for the entry generation is shown in Listing 4 which is generated out of the ePM DSL file shown in Listing 3. The entry ids correspond to the entry type. Entry metadata is predefined by the framework and contained in every entry. The application properties

[8] Taken from `https://github.com/kevinic/texthon/`

section is generated from the DSL with annotated default values. Finally in the header file the prototypes for initialisation and serialisation are created.

In Listing 5 the corresponding source code is listed. Here the initialisation and serialisation functions are implemented. The `SERIALIZE` and `DESERIALIZE` macros refer to a serialisation implementation for primitives implemented by the ANSI C framework.

As for entries, also services, wirings, peers and the topology are generated. To smoothen the usage of the Peer Model Compiler reference implementation also a Makefile is generated which rebuilds files upon source changes.

```
entry digital_in is
  coordination data is
    sensor   : int  := -1;
    state    : bool := false;
end entry digital_in;

entry digital_in_ignore is
  coordination data is
    empty
end entry digital_in_ignore;

entry digital_out is
  coordination data is
    actuator : int  := -1;
    state    : bool := false;
end entry digital_out;

entry ekues_activated is
  coordination data is
    empty
end entry ekues_activated;
```

Listing 3: Example Entry Declaration in ePM DSL. This is the basis for Listing 4 and Listing 5.

```c
/*
 * This file is autogenerated so it won't make any sense to change it!
 */
#ifndef __PM_GENERATED_ENTRIES__
#define __PM_GENERATED_ENTRIES__
#include <PeerModel/entry.h>

typedef uint32_t entry_query_bitset_t;

/* ===== IDs ===== */
typedef enum entry_ids_e {
    ENTRY_ID_INVALID = 0,
    ENTRY_ID_DIGITAL_IN = 1,
    ENTRY_ID_DIGITAL_IN_IGNORE = 2,
    ENTRY_ID_DIGITAL_OUT = 3,
    ENTRY_ID_EKUES_ACTIVATED = 4
} entry_ids_t;

#define ENTRY_ID_MAX            (5)

/* ===== Structs ===== */
typedef STRUCT entry_digital_in_s {
    /* meta data */
    ENTRY_METADATA
    /* user data */
    int32_t sensor; /* default: -1 */
    bool_t state; /* default: FALSE */
} entry_digital_in_t;

/*!
 * This entry is for ignoring digital_in entries for some time.
 */
typedef STRUCT entry_digital_in_ignore_s {
    /* meta data */
    ENTRY_METADATA
} entry_digital_in_ignore_t;
/*** and more ... ***/

/* ===== Initialization Prototypes ===== */
void entry_init_digital_in(peer_address_t origin,
                           /*@out@*/ entry_digital_in_t *s);
uint8_t *entry_serialize_digital_in(const entry_digital_in_t *s,
                           /*@out@*/ uint8_t *dest);
const uint8_t *entry_deserialize_digital_in(const uint8_t *src,
                           /*@out@*/ entry_digital_in_t *s);
/*** and more ... ***/

#endif /* __PM_GENERATED_ENTRIES__ */
```

Listing 4: Entries C Header File Fragment generated out of Listing 3.

```c
/*
 * This file is autogenerated so it won't make any sense to change it!
 */
#include "entries.h"

void entry_init_digital_in(peer_address_t origin, entry_digital_in_t *s)
{
    entry_init_metadata(origin, (entry_t *) s, ENTRY_ID_DIGITAL_IN);
    s->sensor = -1;
    s->state = FALSE;
}

uint8_t *entry_serialize_digital_in(const entry_digital_in_t *s,
                                    uint8_t *dest)
{
    /* serialize metadata */
    SERIALIZE(entry_t, *s, dest);
    SERIALIZE(int32_t, s->sensor, dest);
    SERIALIZE(bool_t, s->state, dest);
    return dest;
}

const uint8_t *entry_deserialize_digital_in(const uint8_t *src,
                                            entry_digital_in_t *s)
{
    /* deserialize metadata */
    DESERIALIZE(entry_t, src, *s);
    DESERIALIZE(int32_t, src, s->sensor);
    DESERIALIZE(bool_t, src, s->state);
    return src;
}

/*** and more ... ***/
```

Listing 5: Entries C Source Code Fragment generated out of Listing 3.

6.3 LaTeX Documentation

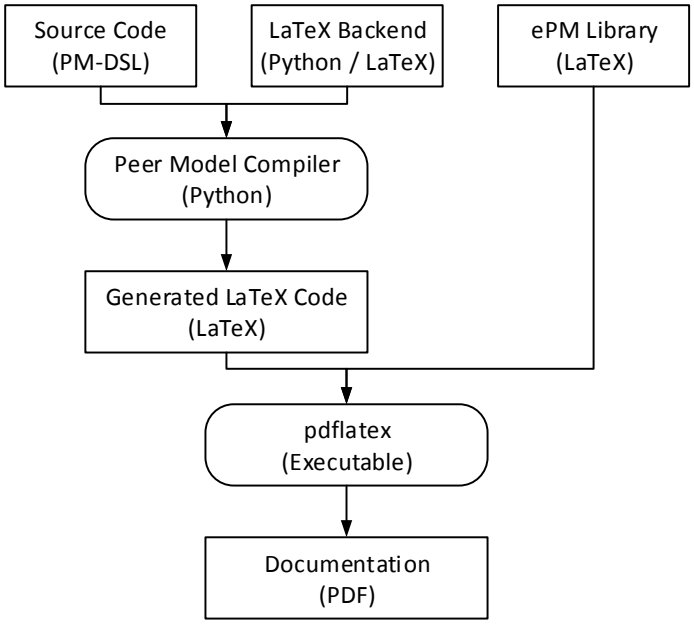


Figure 6.13: The LaTeX Documentation Generation Process.

Software documentation plays a major role in software debugging, reviewing and in understanding and deciphering foreign code. To support developers an automatic documentation generation has been integrated into the toolchain to ease coordination code reviewing during development.

The automatic code documentation is founded on the Peer Model's graphical notation and has been realised in LaTeX with the usage of the PGF/TikZ.

To generate documentation out of the DSL, a similar process like for the executable generation has to be executed: The source code in DSL is assumed to be available. A configuration file has to be created to select the LaTeX output target. Then the Peer Model Compiler has to be executed, producing the LaTeX documentation out of the application source code. Finally the documentation can be translated to Portable Document Format (PDF) by executing "pdflatex". This process is illustrated in Figure 6.13.

The framework includes a set of files containing macros for drawing wirings and peers and a "lstlistings" syntax definition for Peer Model DSL code. The backend generates service code listings, standalone wirings and peers with included wirings.

The generated LaTeX code looks like listed in Listing 6 and its result is shown in Figure 6.14 for standalone wirings and Listing 7 for peers with wirings with the result displayed in Figure 6.15.

```
% ... \documentclass, usepackage ...
\begin{document}
  \begin{center}
  \begin{adjustbox}{width=\textwidth,keepaspectratio}
  \setWiringArrowServiceSpace[\wiringWidth/5]
  \begin{peerless}
    \BeginWiring{forward\_sensor}
      \BeginService{forward\_sensor}
        \upTakeArrow{\textTake ~ sensor}{}{\all}
        \serviceArrowSpace
        \downArrow{sensor\_transport}{}{\all}
        \downArrow{sensor}{}{\all}
        \downArrow{toggle\_red}{}{\all}
      \EndService
      \picRead{sensor\_ack}{}{\none}
      \picTake{sensor}{}{}
      \picAction{sensor}{}{\all}
      \pocAction{sensor\_transport}{}{\all}
      \picAction{toggle\_red}{}{\all}
    \EndWiring
  \end{peerless}
  \end{adjustbox}
  \end{center}
  % ... here more wirings would follow ...
\end{document}
```

Listing 6: Generated LaTeX code for Standalone Wirings.

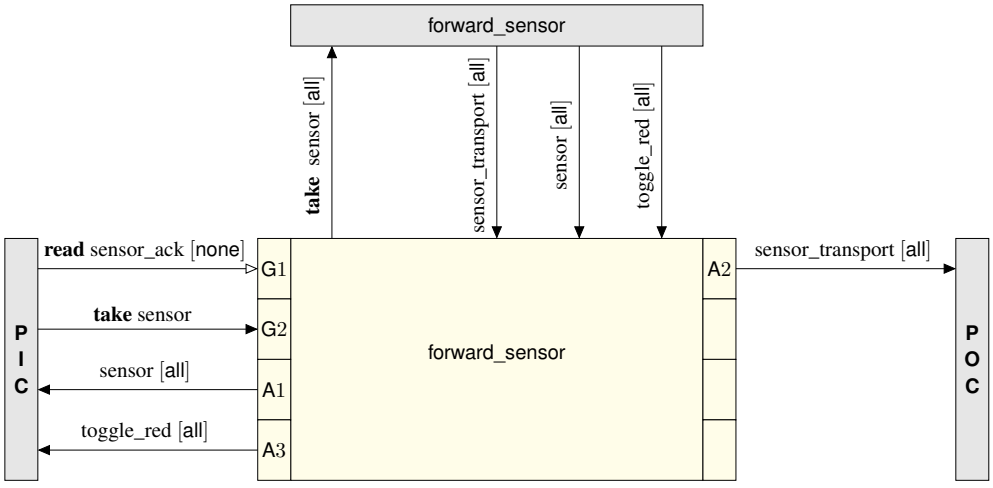


Figure 6.14: Wiring as Listed in Listing 6.

```
% ... \documentclass, usepackage ...
\begin{document}
  \begin{center}
    \begin{adjustbox}{width=\textwidth,keepaspectratio}
    \begin{peer}{forwarder\_ekues}
    \setWiringArrowServiceSpace[\wiringWidth/4]
    \setSlotHeight[2]
    \BeginWiring{forward\_ekues\_activation}
      \BeginService{forward\_ekues\_activation}
        \upTakeArrow{ekues\_activation}{}{\all}
        \serviceArrowSpace
        \downArrow{ekues\_activation}{}{\all}
        \downArrow{toggle\_yellow}{}{\all}
      \EndService
      \setAutoAdjustWiringArrowRight
      \picTake{ekues\_activation}{}{}
      \pocAction{ekues\_activation}{}{\all}
      \picAction{toggle\_yellow}{}{\all}
    \EndWiring
  \end{peer}
  \end{adjustbox}
  \end{center}
  % ... here more wirings would follow ...
\end{document}
```

Listing 7: Generated LaTeX code for a Peer with One Wiring.

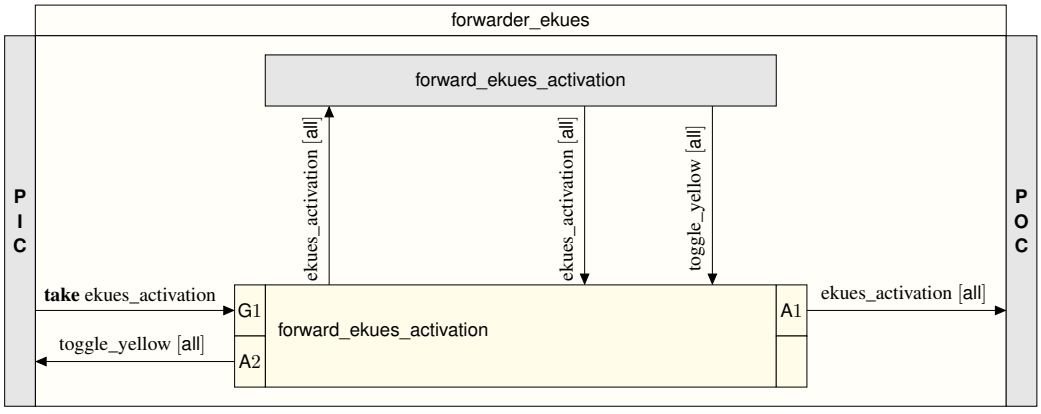


Figure 6.15: Peer with one Wiring as Listed in Listing 7.

CHAPTER 7

Evaluation

The evaluation is carried out by comparing realisations of different use case scenarios regarding selected evaluation criteria. Afterwards the realised toolchain is compared with existing related systems already mentioned in Section 2.5.

7.1 Case Studies

The chosen use cases for the evaluation cover classical MCU applications and some real world scenarios. The flexibility of the reference implementation is benchmarked by implementing a scenario and then the needed modifications to meet changes in requirements are measured, like in [54].

7.1.1 Timed Light Switch

The idea behind this use case is to activate a light when the light switch is pressed and to automatically deactivate it after a specified time. This scenario represents a classical Digital I/O MCU application.

Single Light Switch

The basic implementation is a single timed light switch. This light switch should not be retriggerable, which means that if the button is pressed when the light is on the on-duration will not change.

Remarks The easiest way to implement this on a MCU is to wait for a button press, activate the light, call a delay-routine and deactivate the light afterwards.

Dual Light Switch

After implementing the single light switch the application should be extended to handle two or more light switches with different lights.

Remarks In contrast to the first task this cannot (or only hardly) be implemented with delay routines. For that, the usage of a timer is strongly encouraged.

Retriggerable Light Switch

The next change in requirements is to implement a retriggerable light switch. This implies that when the light is activated and the button is pressed the activated duration is extended.

Remarks This application should only require minimal changes to the previous one, namely removing the check if the light was already triggered upon button press. Although it is easy to adapt this from the 'Dual Light Switch', it can be cumbersome to start with the 'Single Light Switch' as basis.

Distributed Light Switch

To make the light switch scenario more realistic, the light switch (sensor) and the light (actuator) is split up and connected via a wireless network. For this scenario the light switch sends an activation signal to the light and the light handles the automatic turn-off.

7.1.2 Railway Level Crossing Notification

This scenario implements the motivating use case presented in Section 1.1. Parts of this use case have already been described in [45, 42] and are discussed more detailed in this section.

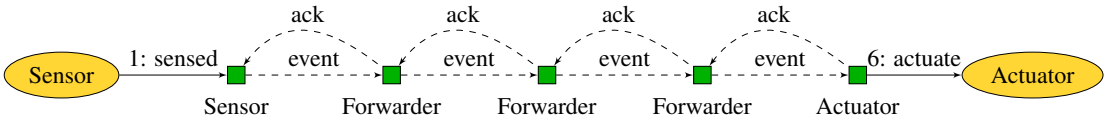


Figure 7.1: Level Crossing Notification.

As shown in Figure 7.1 some nodes are assumed beside a railway track. The first node, namely the sensor node, is connected to a train (wheel) sensor, the last node, called actuator node, is attached to an actuator and intermediate nodes are so-called forwarder nodes which are needed as range extenders. Solid arrows denote wired connections and dashed arrows are wireless connections (which have a potentially higher Packet Error Rate (PER)).

End-to-End Acknowledgement

The first scenario which has to be implemented is an end-to-end acknowledgement, shown in Figure 7.2. Here the event gets first routed to the destination node and is acknowledged by

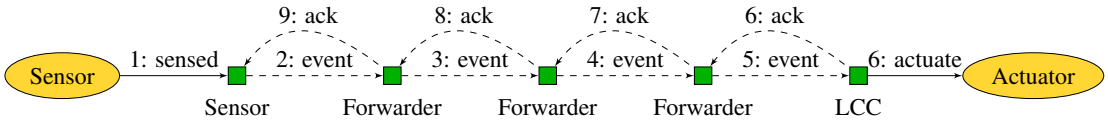


Figure 7.2: End-to-End Acknowledgement.

the final recipient afterwards. When a message gets lost, the message initiator (sensor node) is responsible for retrying message transport.

Point-to-Point Acknowledgement

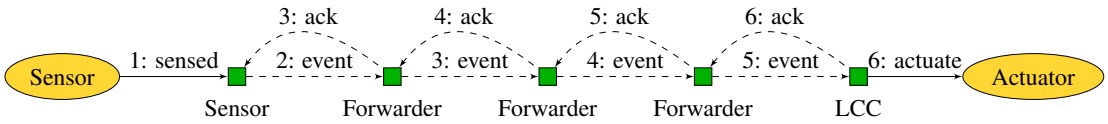


Figure 7.3: Point-to-Point Acknowledgement. [42]

For WSANs with high PERs a point-to-point acknowledgement protocol, which can be seen in Figure 7.3, usually offers faster notification and less messages because packet repetitions are done by the last node on the path which has successfully received the message.

Implicit Point-to-Point Acknowledgement

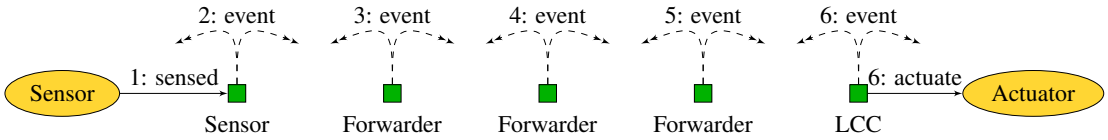


Figure 7.4: Implicit Point-to-Point Acknowledgement.

For the implicit point-to-point acknowledgement, which is sketched in Figure 7.4, instead of unicast transmissions, broadcasts are used for transmitting events. This reduces the message amount since explicit unicast acknowledgement messages are only needed when the implicit acknowledgement is not received at the previous node.

7.1.3 Industrial Automation

For this use case a simple factory which spot welds solder tabs onto button cells. The assumed production line is equipped with belt conveyors, one pick and place and one spot-welding industrial robot. The workflow, as visualised in Figure 7.5, is the following: battery cells and solder tabs are supplied by the vibratory bowl feeder and then transported by the belt conveyors. When one battery and one solder tab arrive at the sensors, the pick-and-place robot moves them to the welding table and are spot-welded afterwards. After welding the battery is delivered by a conveyor to a container for finished batteries. Required emergency-stop mechanisms are out of concern for this application.

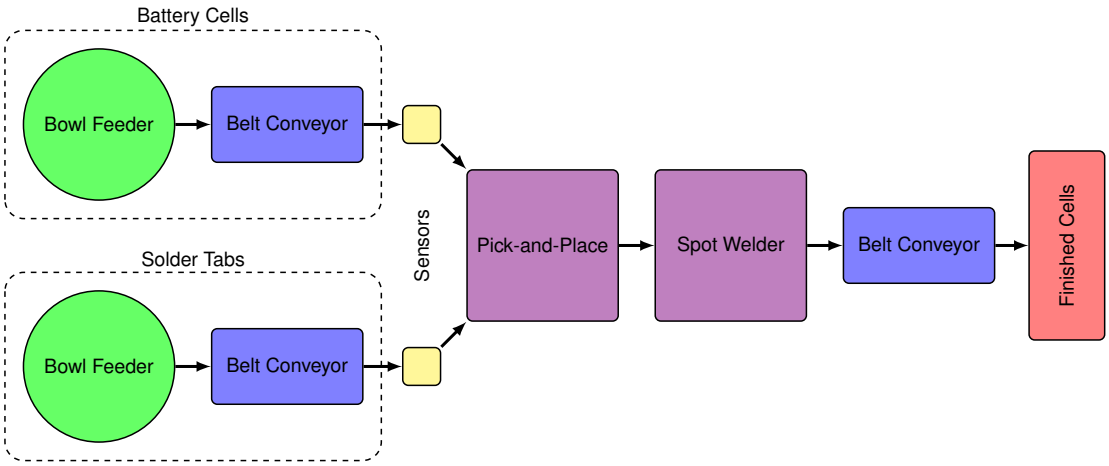


Figure 7.5: Solder Tab Welding Production Line.

Battery and Solder Tab Linear Feeder

The linear feeders provide batteries and solder tabs to the pick-and-place robot. From the bowl feeders batteries and solder tabs can randomly arrive on the right side or upside down. To ensure that the solder tab is welded onto the right side of the battery, a sensor checks if batteries/solder tabs are lying on the right side and that only one battery/solder tab comes at the same time. In case that the object is on the wrong side or there is more than one piece on the conveyor belt, they get blown back into the bowl feeder.

Pick-and-Place

This task utilises the pick-and-place robot arm with a sucker (or gripper) tool. The arm has to take a battery or a solder tab, move it to the welding table and release it again.

Decentralised Automation

The whole application glues the sub tasks together and should activate the feeding bowls until according parts arrive at the corresponding sensors. When each sensor is satisfied, the pick-and-

place robot has to place the objects on the welding table and the spot welding arm combines the battery and the solder tab. Then the finished battery has to be picked and moved to the last belt conveyor which transports the items to a container for finished batteries.

7.2 Implementation

7.2.1 Timed Light Switch

Arduino Implementation

The Arduino implementation of the single timed light switch exercise is listed in Listing 8. Here the obviously simplest way is to implement the timing behaviour by using the provided `delay` function. One drawback of such `delay` routines are that they are usually implemented in a busy-waiting way which consumes more energy because sleep mode usage is prevented. This issue could be solved by implementing the delay routine in an interrupt-driven way. Another problem is that the usage of those delay blocks the program execution and only interrupt-driven application logic can be executed while waiting until the requested time elapsed. This can either be "solved" by utilising an existing multi-threaded OS or by implementing an own scheduler.

```
const int buttonPin = 2;
const int ledPin    = 5;

int buttonState = 0;

void setup()
{
  pinMode(ledPin, OUTPUT);
  pinMode(buttonPin, INPUT_PULLUP);
}

void loop()
{
  buttonState = digitalRead(buttonPin);

  if(buttonState == LOW)
  {
    digitalWrite(ledPin, HIGH);
    delay(30000);
    digitalWrite(ledPin, LOW);
  }
}
```

Listing 8: Arduino Implementation of the Single Light Switch.

In Listing 9 the solution to the dual light switch exercise is shown. To dispatch both light switches the "millis" function is used. This method returns the number of elapsed milliseconds since the start of the application. Upon button press the elapsed milliseconds are stored and the

```
const int buttonPin1 = 2;
const int buttonPin2 = 3;
const int ledPin1    = 5;
const int ledPin2    = 6;
int buttonState1 = 0,
    buttonState2 = 0;
int timer1 = 0,
    timer2 = 0;
bool active1 = false,
     active2 = false;

void setup()
{
  pinMode(buttonPin1, INPUT_PULLUP);
  pinMode(buttonPin2, INPUT_PULLUP);
  pinMode(ledPin1, OUTPUT);
  pinMode(ledPin2, OUTPUT);
}

void loop()
{
  buttonState1 = digitalRead(buttonPin1);
  buttonState2 = digitalRead(buttonPin2);
  if(active1)
  {
    if(millis() - timer1 >= 30000UL)
    {
      digitalWrite(ledPin1, LOW);
      active1 = false;
    }
  }
  else if(buttonState1 == LOW)
  {
    digitalWrite(ledPin1, HIGH);
    timer1  = millis();
    active1 = true;
  }
  if(active2)
  {
    if(millis() - timer2 >= 30000UL)
    {
      digitalWrite(ledPin2, LOW);
      active2 = false;
    }
  }
  else if(buttonState2 == LOW) {
    digitalWrite(ledPin2, HIGH);
    timer2  = millis();
    active2 = true;
  }
}
```

Listing 9: Arduino Implementation of the Dual Light Switch.

light is activated, when the difference with the current milliseconds and the stored value is above 30 secs the light is reset again.

As it can be seen clearly, a small change in requirements can result in a complete code rewrite when no scheduling mechanisms are present.

To implement the retriggerable light switch the `else if` condition should be replaced by a normal `if` condition.

Embedded Peer Model Implementation

The ePM implementation of the single light switch can be seen in Listing 10 and the corresponding graphical documentation output is shown in Figure 7.6.

The only required wiring, namely "activate_light", is triggered upon a "digital_in" entry from sensor 1 (which equals to the first button) with a set state (which means that the button is pressed). Then the service "activate_light", emitting a "digital_out" entry to activate the LED output and another "digital_out" entry turning of the light after 30 secs, is executed. To ensure that the light is turned off after 30 secs, the TTS coordination property is used. After executing the service, the wiring's action block delivers the LED request to the POC. A re-triggering is disabled by the "none" query on the "digital_out" guard.

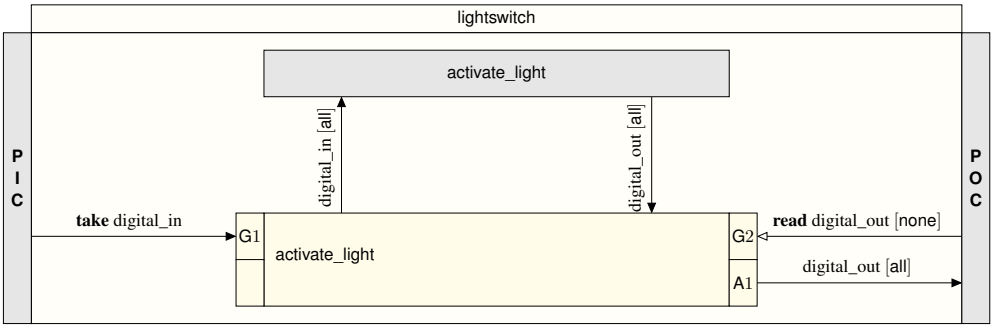


Figure 7.6: Single Light Switch Application. Queries are omitted.

For the double light switch example, only the condition `(digital_in.sensor == 1)` has to be removed from the `digital_in` and the condition `[[digital_out.actuator == digital_in.sensor]]` has to be added to the `digital_out` guard link.

For the retriggerable light switch instead of the "none" query on the "digital_out" guard, an "all" selection has to be used.

7.2.2 Railway Level Crossing Notification

The graphical representations of the peer implementations for the "End-to-End" and "Point-to-Point" scenario are shown in Figure 7.7, Figure 7.8 and Figure 7.9. At first the "sensor" peer waits for a wheel sensor event by executing the wiring "sensor" (Figure 7.7) which triggers upon a "`digital_in`" entry arrival. Then it emits an entry "sensor", demanding the reliable transmission of the sensor event. The emitted entry also gets a new FLOW identifier. Whenever a "sensor" entry is available in the PIC and its TTS elapsed, it is taken by the "send_sensor"

```
use library digital_io;

service activate_light is
    take digital_in[all];
    emit digital_out[all];
begin
    create(digital_out)
        .set(state => true, actuator => digital_in.sensor)  .emit
        .set(state => false, tts => now +30s)               .emit;
end service activate_light;

wiring activate_light is
begin
    take digital_in [[ (digital_in.state) and (digital_in.sensor == 1) ]];
    read digital_out[none] suppress tts from POC;
    running service activate_light;
    deliver digital_out[all];
end wiring activate_light;

peer lightswitch is
begin
    instantiate wiring activate_light;
end peer lightswitch;

topology example is
begin
    processor lightswitch runs peer lightswitch;
end topology example;
```

Listing 10: Peer Model Implementation of the Single Light Switch.

wiring and a “sensor_transport” is sent to the next node in the chain. Additionally a new “sensor” entry with a TTS of the retry time is placed in the PIC. In case of the “Point-to-Point” scenario the next forwarder node, and in case if the “End-to-End” setting the LCC node accepts this “sensor_transport” entry with the wiring “accept_sensor”. This wiring then sends an “ack” entry, acknowledging the received sensor event, to the last node (either the sensor or forwarder node) and saves a “sensor” entry registering that an event occurred. Please note that the “take sensor[all]” link, which is executed with TTS suppression, ensures that there is not more than one “sensor” entry from the same sensor event in the PIC. The next node starts the same send-retry logic (by using the same wiring logic) as the node before. Going back to the “sensor” peer, here an “ack” entry should have arrived which triggers the “acknowledge_sensor” wiring, cleaning up the corresponding “sensor” (with TTS suppression), “sensor_transport” and “ack” entries. In case any entry is lost on the air channel the retry mechanism will ensure that the message is transmitted again. This described procedure is executed until the sensor event finally arrives at the “levelcrossing” peer, representing the LCC, which only accepts the “sensor” entry and acknowledges its reception.

For all the guard links used in this application the FLOW correlation is used to enable dis-

tinction of different sensor events.

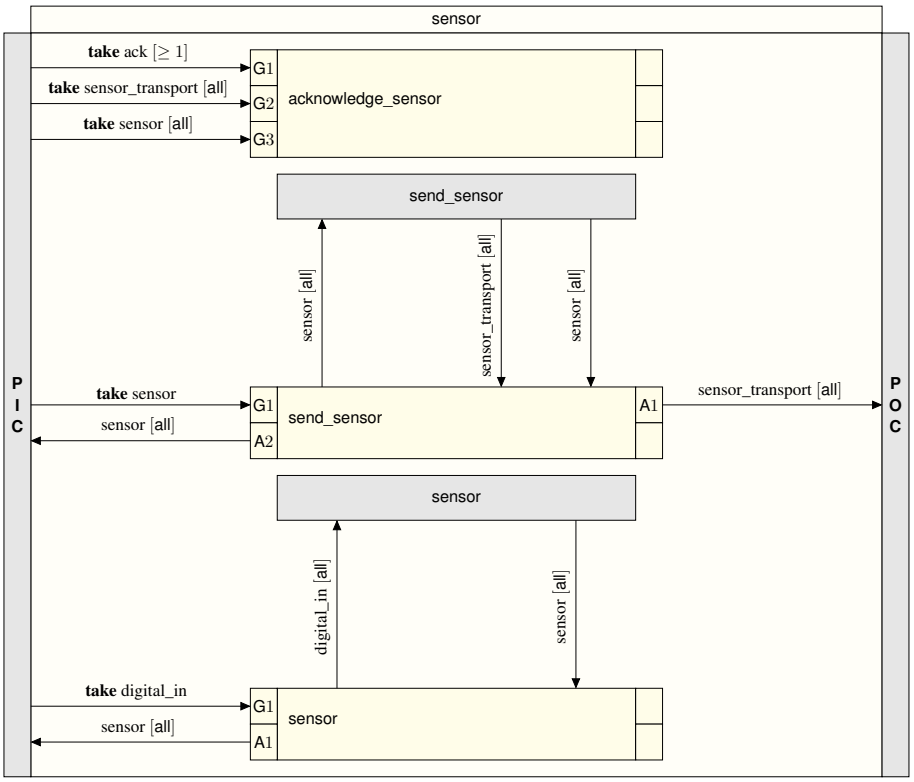


Figure 7.7: Sensor Peer used in the “End-to-End” and “Point-to-Point” scenarios.

The “Implicit Point-to-Point” basically uses the same logic, but it sends the “sensor_transport” entry with a broadcast instead of an unicast (by setting DEST to `ANY`) and attaches a property, called “from_node”, where this “sensor” entry came from. Additionally instead of the “ack” entry, a “sensor” entry with the query `[[my_nr < sensor_transport.from_node]]`, is requested. This query determines if the received entry is meant as an “ack” entry (then the “sensor” comes from a node nearer to the LCC node).

7.2.3 Industrial Automation

The industrial automation applications are rather simple wirings and their graphical representation is not printed.

Linear Feeder All belt conveyors are handled by one wiring which activates the linear feeder for a certain time by using the TTS property upon a feeder sensor event occurred. The sensors for blowing out wrong parts has a similar behaviour but a different time how long the air flow is activated.

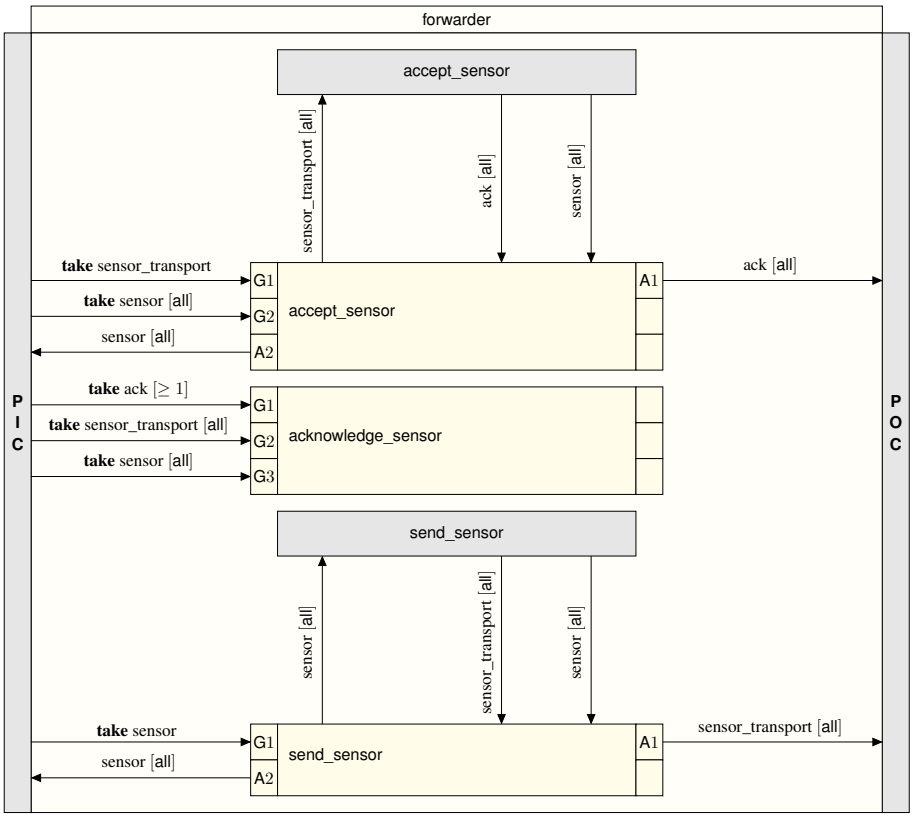


Figure 7.8: Forwarder Peer used in the “Point-to-Point” scenarios.

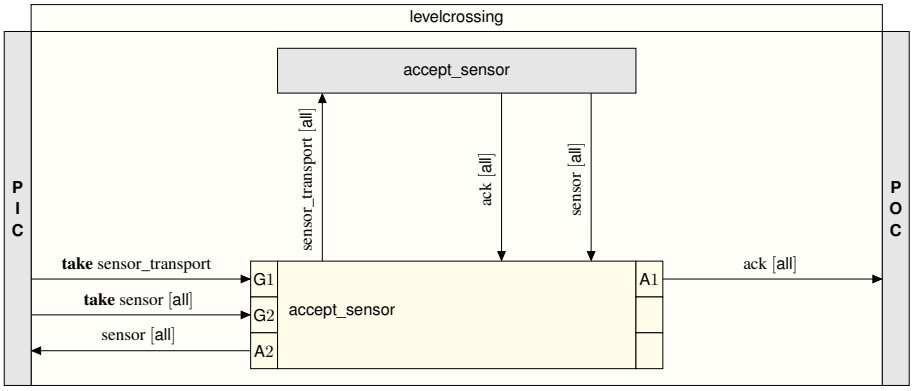


Figure 7.9: LCC Peer used in the “End-to-End” and “Point-to-Point” scenarios.

Pick-and-Place The pick and place is also covered by one wiring which enables the robot sequence upon a sensor event as follows: At first the pick position has to be reached by the robot arm, then the sucker/gripper tool has to be activated, afterwards the target position has to be reached and finally the sucker/gripper tool has to release the item. This is also realised with the TTS property.

Decentralised Automation The spot welder arm executes nearly the same procedure as the pick-and-place arm and the bowl feeders have a similar behaviour like the linear feeder.

Finally the application is glued together with one topology file.

7.3 Measurement Setup and Results

Source Code Size

The evaluation of source code size should give a rough estimation about the time spent for realising the application. For sure this measure does not take debugging efforts into account. Source code sizes are measured by the line count, ignoring empty and comment lines and libraries are not included in this measure.

The difference of line counts is evaluated and listed in the Δ_{LoC} column. Additionally also the effective line difference is evaluated with a `diff`-tool and listed in the Δ_{eff} column. The effective line difference denotes the number of lines which are different in both files.

In Table 7.1 the specific results of the source code evaluation are listed. Especially for the light switch scenario, the amount of reusable code (represented by Δ_{eff}) is very large compared to the native Arduino implementation. When adapting the light switch to a distributed version, both systems need a rather big code change. However having the logic of multiple nodes in one topology increases the code re-usability and makes it easier to keep a clear view over the whole project.

The railway scenario also demonstrates the high re-usability for similar tasks. Even adding/removing further nodes would only result in a small source code change.

For the industrial automation the source code differences are not applicable since they cover different tasks (and not a similar task with rising difficulty), although they are listed for the sake of completeness.

Binary Size

The easiest way to evaluate the binary size is by using according "`size`" tool from the `binutils` package. To get comparable values, the source code has to be compiled with similar compile flags and optimisation levels. In contrast to the source code size, the libraries are included in the binary sizes.

Since the Arduino environment enforces some optimisation flags, like "`-Os`" (optimisation regarding size), these flags have also been adopted for the Peer Model ANSI C reference implementation.

Application	Scenario	Nodes	Arduino			ePM		
			LoC	Δ_{LoC}	Δ_{eff}	LoC	Δ_{LoC}	Δ_{eff}
Light Switch	Single Light Switch	1	18			24		
	Dual Light Switch	1	49	+31	±43	24	+0	±1
	Retriggerable Light Switch	1	49	+0	±2	24	+0	±1
	Distributed Light Switch	2	81	+32	±55	46	+22	±29
Railway Track	End-to-End Ack.	5	137			73		
	Point-to-Point Ack.	5	177	+40	+47	82	+9	±12
	Implicit P2P Ack.	5	196	+19	+27	76	−6	±18
Industrial Automation	Linear Feeder	2	49			35		
	Pick-and-place	+2	30	n/a	n/a	26	n/a	n/a
	Decentralised automation	+2	52	n/a	n/a	23	n/a	n/a

Table 7.1: Source Code Size Comparison measured in Lines-of-Code.

Please note that the distributed scenarios are not taken into account since therefore the total or an average value of all nodes has to be used which is not comparable to single-node applications.

Application	Scenario	Nodes	Arduino			ePM		
			Bytes	Δ		Bytes	Δ	
Light Switch	Single Light Switch	1	1 138			9 484		
	Dual Light Switch	1	1 212	+75	+7%	9 228	−256	+3%
	Retriggerable Light Switch	1	1 208	−4	+0%	9 448	+110	+2%

Table 7.2: Binary Output Size Comparison for the Arduino Platform.

In Table 7.2 the resulting binary sizes of the corresponding applications are listed. Compared to the native Arduino framework, the ePM implementation has a rather high binary size. This overhead partially comes from the framework and HAL implementation. The estimated library overhead is therefore estimated with less than 10 kB for the Arduino ePM implementation.

Energy Consumption without Wireless Radio

The energy consumption was measured with the Advanced Energy Monitoring (AEM) [21] by EnergyMicro[1] attached to the target platform. The AEM is a specialised voltage and current measurement circuit for low-power, low-current applications. A screenshot of a measurement where two LEDs are alternatingly activated can be seen in Figure 7.10.

To only measure the MCU's power consumption, external peripherals, like LEDs and wireless transceivers, have been removed for current measurements. Whenever an user input is required for the application, this is simulated by an external device simulating periodic button presses. For the light switch scenario about the same duration as the light is turned on is assumed for the pause between successive simulated button presses.

The comparison of the reference implementation's current consumption is summarised in Table 7.3. Since the used application only has a minor impact on the current consumption, namely only in how often the micro-controller is woken up, and these wake up intervals are typically

[1] Taken from the "EFM32GG-STK3700" starter kit.

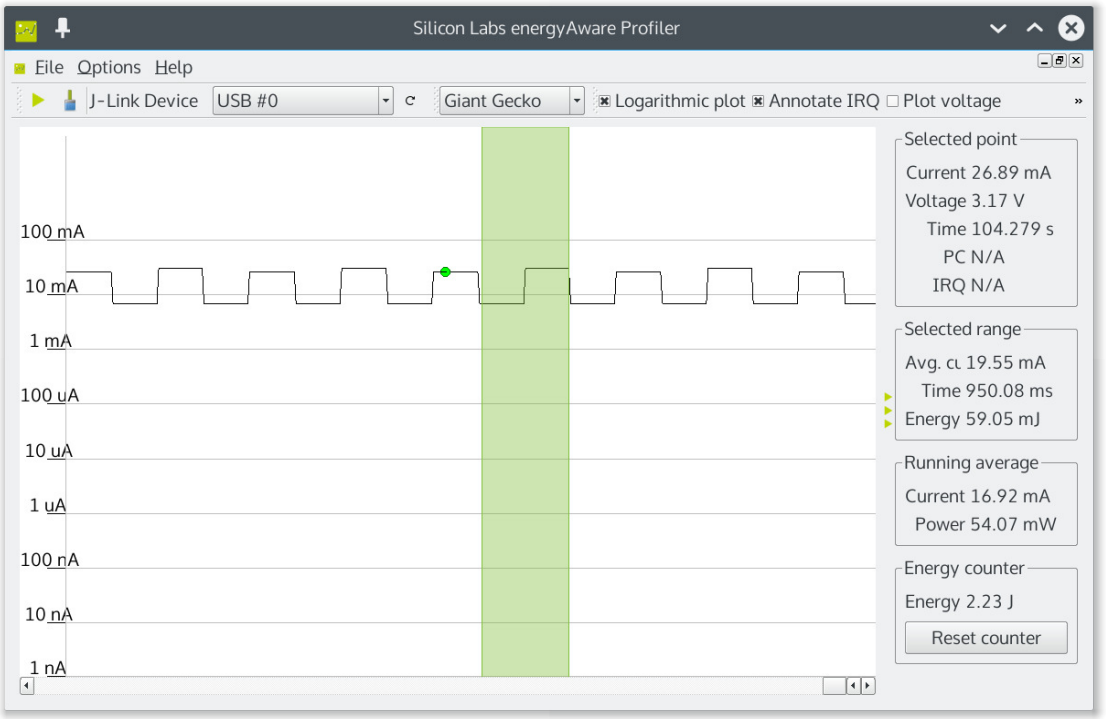


Figure 7.10: Example of an AEM Measurement. The measured application activates a green LED for 500 ms (marked with a green circle), then LEDs are deactivated for 500 ms, then a red LED is activated for 500 ms and then the LEDs are turned off again for 500 ms.

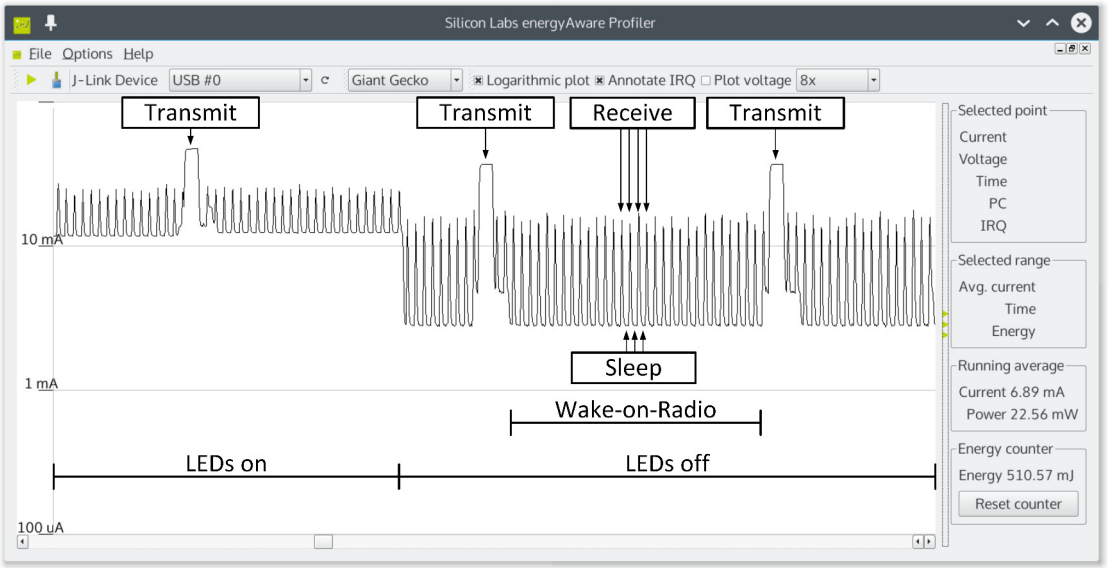


Figure 7.11: Current Measurement of a LOPONODE with a Wireless Transceiver.

very short, the values are summarised for the whole scenarios. As it can be seen, the current consumption is slightly lower for nodes using a sleep mode and the specialised LOPONODE hardware uses less energy than the chosen Arduino board.

Application	Arduino [mA]	ePM	
		Arduino [mA]	LOPONODE [mA]
Light Switch	8.15	6.97	2.38
Railway Track	–	6.97	2.36

Table 7.3: Current Consumption Comparison for all Platforms. The voltage is fixed to 3.3 V.

Energy Consumption with Wireless Radio

A measurement of a LOPONODE with a CC1101 wireless transceiver attached, periodically sending a message every 500 ms can be seen in Figure 7.11. The wireless transceiver is configured for 433 MHz and an transmission output power of 10 mW. Additionally a technique called "Wake-on-Radio"[2] is used, which puts the radio in a sleep mode and periodically wakes it up and listens for a correct preamble. That mechanisms reduces the power consumption depending on how long the sleep mode is entered and how long it listens for a valid preamble but the longer the sleep modes are chosen the more packets get lost. In Figure 7.11 the wake-up instants are represented by spikes in current consumption. The average energy consumption over a long time measurement is about 8 mA, strongly depending on the packet size, message sending frequency and "Wake-on-Radio" configuration.

In this example it can clearly be seen that in difference to the current consumption without a wireless radio (less than 3 mA), the major role in terms of power consumption for WSAN nodes plays the used wireless module and its MAC layer. The MAC layer has a high impact on the wireless radio's consumed energy because it determines the amount of transmit, receive and sleep operations.

Fieldtest

To demonstrate the capableness and get an estimation of the availability of the LOPONODE link, field tests along a railway track have been carried out. These field tests were located between Weikendorf and Stripfing near Gänserndorf, which can be seen on the map in Figure 7.12. Boxed LOPONODEs with 2000 mAh batteries but without solar cells have been placed at the marked points. An impression of the field test setup is given in Figure 7.13, Figure 7.14 and Figure 7.15. The applications under test were the railway use case implementations "Point-to-Point Acknowledgement" and "Implicit Point-to-Point Acknowledgement" as shown in the implementation section.

For all fieldtests, the used hardware was a LOPONODE with an integrated 2000 mAh battery. Instead of a real wheel sensor an external sensor simulator, triggering a train event every 30 secs, was attached to the first LOPONODE.

[2] In general this energy saving method is called "Low Power Listening".

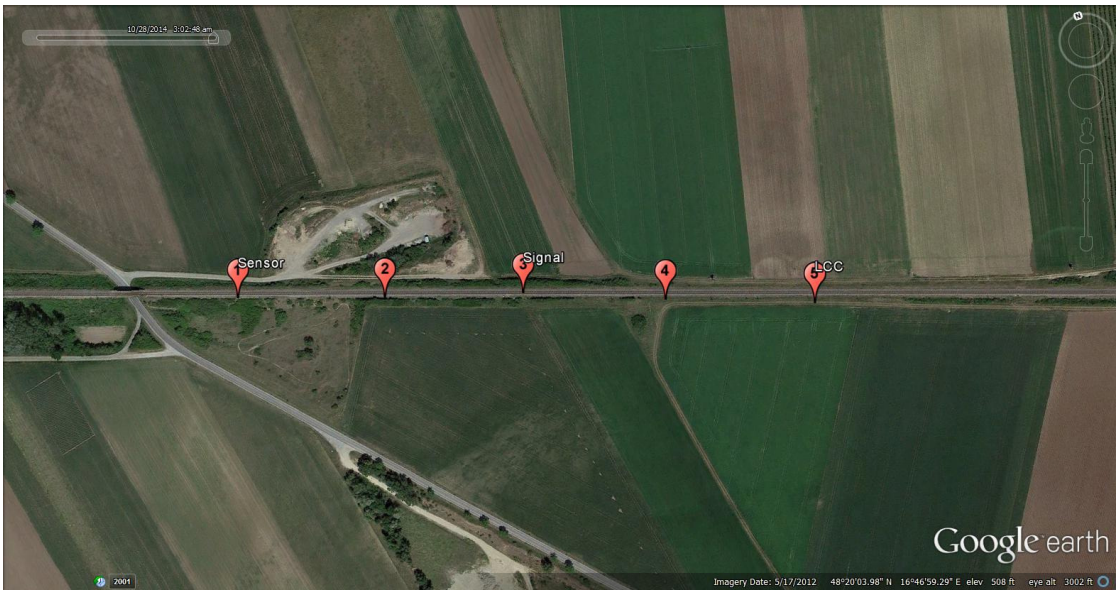


Figure 7.12: LOPONODE Fieldtest Setup near Gänserndorf. Image exported from Google Earth.

Figure 7.13: Fieldtest Setup Impressions: Railway Track Situation. © by eva Kühn.

Figure 7.14: Fieldtest Setup Impressions: Nodes mounted on Telegraph Poles. © by eva Kühn.

Figure 7.15: Fieldtest Setup Impressions: Team. © by eva Kühn.

In the first fieldtest – happening from 28th of October to 3rd of November, 2014 – the "Point-to-Point Acknowledgement" application was tested. An interesting fact we observed was that whenever a real train passed by, the storage cards stopped working for some time. This was caused by loose mechanical contacts in the microSD card slot and was fixed with a mechanical support attached to the storage cards for the following fieldtests. One result of this fieldtest was an estimation of the run-time of 14 days with one battery charge.

From the 10th of November to the 25th of November, 2014 the second fieldtests took place. The application under test was the same as in the first fieldtest but with improved logging and the duration was chosen based on the estimated run-time with one battery charge. Here first representative results of the Point-to-Point latencies, End-to-End latencies, Received Signal Strength Indication (RSSI) and Link Quality Identifier (LQI) were recorded.

Finally, the third fieldtests happened from 9th of December, 2014 to the 12th of January, 2015. For these test run the "Implicit Point-to-Point Acknowledgement" application was deployed.

Fieldtest	**Measurement**	**Average**	**Standard Deviation**
Second Fieldtest	RSSI	$-78.86\,\mathrm{dBm}$	± 0.63
	LQI	46.79	± 0.56
	Point-to-Point Latency	$81.46\,\mathrm{ms}$	± 58.54
	End-to-End Latency	$540.40\,\mathrm{ms}$	± 85.09
Third Fieldtest	RSSI	$-82.70\,\mathrm{dBm}$	± 2.64
	LQI	47.78	± 1.06
	Point-to-Point Latency	$86.13\,\mathrm{ms}$	± 58.86
	End-to-End Latency	$361.47\,\mathrm{ms}$	± 209.70

Table 7.4: Fieldtest Results.

The results of the second and third fieldtest are shown in Table 7.4. The RSSI value indicates the strength of the received signal. This value strongly depends on the distance between nodes and effective output power. Since all of the nodes work have the same output power configuration and the distance between nodes is about 120 m, this value is very similar for all nodes.

The LQI gives an estimation how good the signal quality is. This depends on effects like interferences. This value seems to be very stable as well which states that environmental changes or bypassing trains only slightly affect the signal quality.

The Point-to-Point Latency gives an estimation of the time needed to transmit a message between a pair of nodes. For the third fieldtest this value is slightly higher because of a higher processing overhead on the receiver side.

Finally, the End-to-End Latency states how long the transmission from the sensor node to the LCC node needs. By using the "Implicit Point-to-Point Acknowledgement" application in the third fieldtests, this value got a lot better coming from less packets on air, improving the signal quality and resulting in less re-transmissions.

During the fieldtests, all sensed train events were successfully propagated through the net-

work and the fieldtests have successfully shown that the realised use case implementations "Point-to-Point Acknowledgement" and "Implicit Point-to-Point Acknowledgement" are working properly.

Summary

The evaluation shows that the code re-usability is increased for distributed scenarios when using the ePM DSL. The energy consumption of the MCU is also very promising and the documentation helps a lot in debugging and visualising developed software.

The library overhead may be an important factor and should be related to the available memory for the according platform: For the Arduino ePM implementation, the framework needs less than 10 kB ROM and the MCU has a ROM size of 32 kB, resulting in a ROM consumption of less than 31 %. The LOPONODE implementation needs less than 20 kB and the LOPONODE hardware offers a MCU size of 1024 kB which results in a ROM usage of less than 2 %.

7.4 Comparison with Related Work

Feature	Systems			
	TCMote	TeenyLIME	PEIS	ePM
Space-based coordination	✓	✓	✓	✓
Rule-based coordination	not available	not available	not available	wiring semantics
DSL	✓	×	×	✓
Code Generation	✓	×	×	✓
Graphical Notation	×	×	×	✓
Low-Power Mechanisms	?	✓	?	✓
Semi-automated Networks	✓	✓	✓	✓
Automated WSAN	×	✓	?	✓
Network Topology	Hierarchical	Peer-to-Peer	Peer-to-Peer	Peer-to-Peer
Supported Platforms	TinyOS	TinyOS	TinyOS	Arduino, LOPONODE

Table 7.5: Feature Matrix of Compared Systems with the ePM.

Compared to related systems the ePM offers higher expressiveness by the introduction of the wiring semantics. Additionally the separation of concerns (coordination tasks and user services) is done in a stricter way. From the analysed systems only TCMote offered a DSL, which essentially embeds into C, and can produce code output. The ePM uses a newly invented DSL, in a declarative way, to enable AVOPT in the future. Code Generation for embedded platforms is realised for the Arduino and LOPONODE platform. In addition to these platforms, an implementation has already been realised for Dynamic C (for the Rabbit RCM4300) [38] and a realisation for C#/.net has been developed in [58]. Further implementations based on Java and Go are currently in development as well. The Peer Model also offers a graphical representation for source code written in the DSL and additionally a visualisation based on log files and structure files created out of DSL is covered [13]. Because of the asynchronous programming model,

energy-efficient operation is supported due to appropriate sleep mode. Heterogeneous systems are supported because of the existence of mentioned implementations above by using a serial bridge which transforms entries serialised in the embedded format to entries of the .net style. Homogeneous, automated WSANs are supported since each node may incorporate any arbitrary application. From the network topology point of view, the ePM primarily supports Peer-to-Peer communication but any other routing may be integrated either on the application or directly in the HAL. The supported platforms differ entirely from related systems. To reach the desired energy efficiency, an own system tailored to the ePM's needs with a very flexible HAL has been realised. This comparison is summarised in Table 7.5.

The applicability of the ePM toolchain has been proven by realising a real-world use case, namely the scenario defined in Section 1.1, and the corresponding promising fieldtests shown in Section 7.3.

CHAPTER 8

Future Outlook

Performance Measurements

Because of missing measuring equipment the performance of the ePM implementation could not be evaluated. For our application where energy-awareness was the most crucial aspect, performance was not a significant factor but for other use cases it should be analysed.

Pattern Composition

Integrating a pattern concept greatly improves code re-usability and enables software composition. For the Peer Model a pattern concept has been developed in [61] and [43] which should be integrated in the DSL.

Integration of eXVSM

Currently, an implementation of XVSM for embedded devices, called "eXVSM" is developed at the Institute of Computer Languages. This eXVSM implementation shall then be integrated into the ePM containers, which should increase the query expressiveness and allows different container coordinators.

Security

Since security is very important for practical applicability and the used LOPONODE hardware has AES support in hardware, a message signing and authentication with Message Authentication Codes could be realised. Even a simplified version of the XVSM security concept, shown in [12], which proposes a role based access control for containers, could be integrated.

Software Deployment

For WSNs with higher bandwidth, for example when using 2.4 GHz transceivers, and scenarios without safety impacts, a Firmware Over-the-Air (FOTA) upgrade mechanism and a Management protocol could be integrated. Before including those mechanisms, the security concept should be implemented to prevent the WSAN from malicious software updates.

Simulation and Verification

Further research should be put into simulation and verification parts of the toolchain. For prototypical simulation environment, a modified version of the ANSI C implementation could serve as a basis. To improve simulation for WSANs the implementation could be integrated into a network simulator as well.

For software verification different approaches, like classical or probabilistic model checking, should be investigated. With some restrictions on the DSL maybe automated software verification could be leveraged for a certain kind of coordination problems.

Programmable Logic Controller Languages

In the domain of Programmable Logic Controllers (PLCs) some DSLs for software development exist. For distributed applications the IEC 61499 defines function blocks with in- and outputs which encapsulates IEC 61131 software.

Since these function blocks are similar to the Peer Model's wiring semantics, a model transformation might be realised.

LOPONODE v2

At the time of finalising this thesis, a new version of the LOPONODE hardware was produced. This new version, namely LOPONODE v2, has been designed with improved Digital I/O capabilities, a different microSD card slot and a variety of new wireless modules complying with it. The newly available wireless modules enable further research and development on LLC and MAC layers, especially in combination with the higher programming abstraction. The new LOPONODE generation can be seen in Figure 8.1 and a set of new wireless modules is shown in Figure 8.2. Please note that the battery is mounted below the board.

Further Platforms

A project called "flutter" [1] is now in its final phase, targeting the production of ARM-based Arduino compatible WSAN nodes for less than €20. To integrate these WSAN nodes, a HAL adaption should be implemented for the ARM based Arduino platform.

[1] `http://www.flutterwireless.com/`

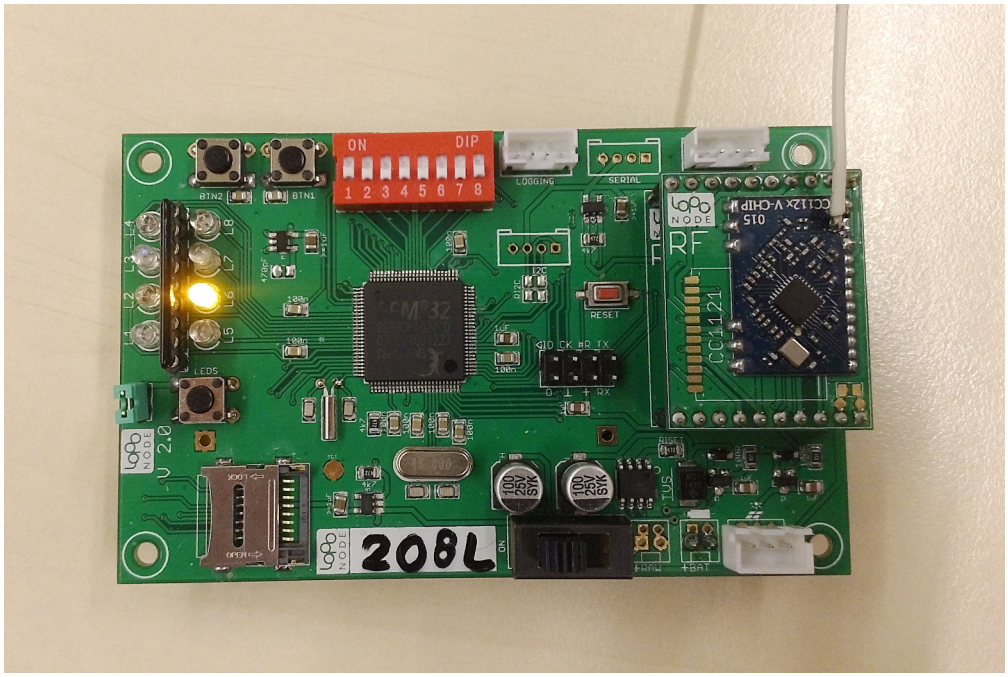


Figure 8.1: LOPONODE v2 Platform with a CC1121 Wireless Radio.

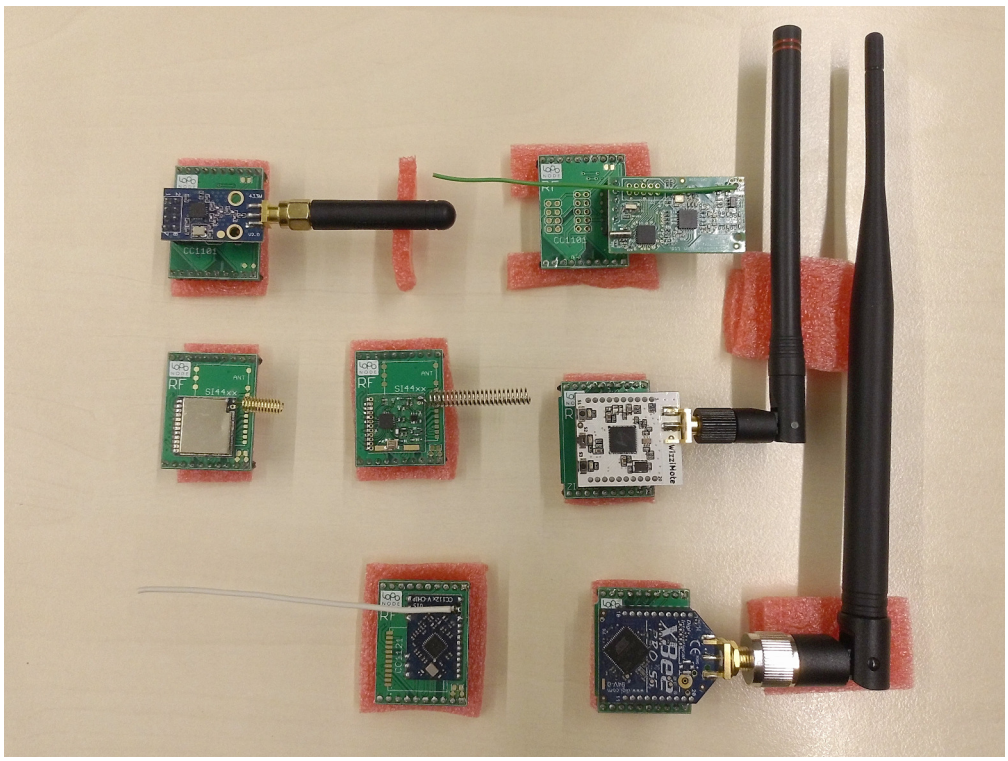


Figure 8.2: Various LOPONODE v2 Wireless Radios.

Hard Real-Time Constraints

For safety-critical appliances, the ePM should be analysed how a notion of real-time constraints could be integrated. Then an implementation based on a Real Time OS (like SafeRTOS[2]) with an automated Worst Case Execution Time analysis could be realised to meet certification requirements.

[2] http://www.freertos.org/FreeRTOS-Plus/Safety_Critical_Certified/SafeRTOS.shtml

CHAPTER 9

Conclusion

The goal of this work was to implement the Peer Model, a space-based programming model, for embedded wireless nodes and to introduce software engineering methods in the domain of distributed embedded systems. The Peer Model targets the software development for distributed and concurrent systems and since WSAN nodes have little memory, CPU power and miss some important features like a MMU, the model was restricted in some points to meet those requirements.

Then a DSL, playing the central role for the toolchain sketched in Figure 9.1, was specified.

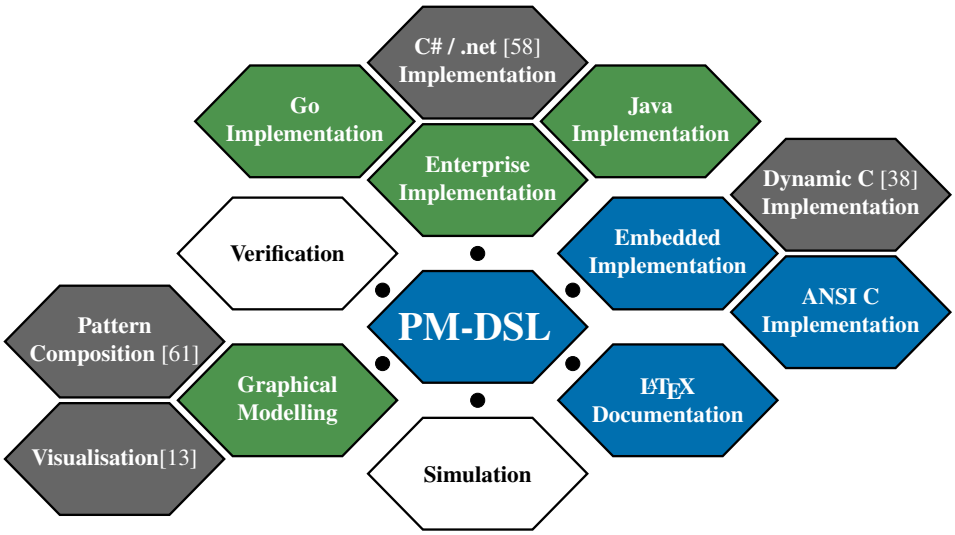


Figure 9.1: Peer Model Toolchain – A Retrospective. Blue parts were covered in this work, green parts are work in progress, grey parts are available from other works, and white ones are planned.

To bridge the gap between design and implementation a code generator for ANSI C was implemented. This code generation bases on an ANSI C framework which incorporates a HAL to enable easy porting to any embedded platform. The ANSI C HAL was realised for two platforms, namely for one Arduino based platform and for a specialised hardware design tailored to the motivating use case's needs.

To get an estimate about energy efficiency and the overhead coming from the ePM framework, the evaluation was carried out by implementing several case studies. The easiest task was realised for a native Arduino and the Arduino ePM implementation. Then a distributed case study, originating from a use case in the railway telematics domain, was implemented with the ePM DSL implementation and benchmarked regarding source code changes for changing requirements. Additionally to the theoretical implementation fieldtests beside a real railway track were performed and gained information about transmission times and energy consumption was evaluated. Finally a decentralised automation exercise was implemented with the ePM DSL and analysed with respect to code reuse.

The results gained from the evaluation showed a high code reusability for scenarios with changing requirements. For the Arduino platform, the energy consumption was reduced by 14 % compared to the native implementation. The specialised LOPONODE platform further improved it by 65 % resulting in a current consumption of 2.38 mA[1]. Nevertheless, these improvements imply in higher binary sizes. The framework overhead for the Arduino platform is about 31 % and about 2 % for the LOPONODE platform.

During the work on the thesis also preliminary results were published in [42] and other works used parts of the developed toolchain:

C# / .net A backend for generating C#/.net source code based on the framework developed in [58] was created.

Visualisation A backend for generating the Visualisation Intermediate Language (VIL) used in [13] was implemented.

Pattern Composition Parts of the LaTeX were used for visualising wirings and peers in [61].

Dynamic C In [38] a backend for "Dynamic C", the programming language used for the Rabbit RCM4300, was realised on the basis of an intermediate version of the ANSI C framework.

An enterprise implementation for Java and Go are currently in development.

[1] This equals to a power consumption of about 8 mW

Glossary

AEM Advanced Energy Monitoring. 78, 79

AES Advanced Encryption Standard. 38, 87

AST Abstract Syntax Tree. 43–45

AVOPT Analysis, Verification, Optimisation, Parallelisation and Transformation. 13, 30, 84

CFG Context-Free Grammar. 44

CISC Complex Instruction Set Computer. 55

COTS Commercial Off The Shelf. 4, 9, 39

CPU Central Processing Unit. 8, 9, 46, 55, 91

DEST Destination. 23–25, 49, 55, 75

Digital I/O Digital Input/Output. 39, 47, 51, 55, 57, 67, 88

DMIPS Dhrystone Million Instructions Per Second. 46

DSL Domain Specific Language. 2–5, 12, 13, 15, 19, 29–31, 43, 44, 47, 49, 58–60, 63, 84, 87, 88, 91, 92

EBNF Extended Backus-Naur Form. 29–31

ePM Embedded Peer Model. 4, 5, 21–26, 29–34, 43, 45, 47–51, 56–58, 60, 73, 78, 80, 84, 85, 87, 90, 92

FIFO First In First Out. 11

FOTA Firmware Over-the-Air. 88

GPL General Purpose Language. 12

HAL Hardware Abstraction Layer. 1, 47, 49, 51, 54, 55, 57, 78, 85, 88, 92

IC Integrated Circuit. 39

IDE Integrated Development Environment. 1

ISM Band Industrial, Scientific and Medical Band. 9, 40

ISO International Organization for Standardization. 40, 56

JTAG Joint Test Action Group. 1

Layer 1 Physical Layer. 40, 56

Layer 2 Data Link Layer. 40, 56

Layer 3 Network Layer. 56

Layer 4 Transport Layer. 56

LCC Level Crossing Controller. 3, 69, 74–76, 83

LED Light Emitting Diode. 32, 39, 41, 73, 78, 79

LiPo Lithium Polymer. 37, 39

LLC Logical Link Control. 56, 58, 88

LOPONODE Low-Power Node. 4, 5, 39, 40, 57, 58, 79–81, 84, 87–89, 92

LQI Link Quality Identifier. 83

MA Mobile Agent. 14–16

MAC Media Access Control. 56, 58, 80, 88

MCU Micro Controller Unit. 1, 4, 9, 10, 14, 18, 21, 22, 26, 38, 39, 45, 47, 50, 51, 54–56, 58, 67, 78, 84

MMU Memory Management Unit. 47, 54, 58, 91

MPU Memory Protection Unit. 58

OS Operating System. 1, 10, 71, 90

OSI Open Systems Interconnection model. 40, 56

PC Personal Computer. 7, 18, 20

PCB Printed Circuit Board. 39, 40

PDA Personal Digital Assistant. 14, 18, 20

PDF Portable Document Format. 63

PEG Parsing Expression Grammar. 44

PER Packet Error Rate. 68, 69

PIC Peer-In-Container. 21, 24–26, 34, 50, 51, 54, 55, 73, 74

PLC Programmable Logic Controller. 88

POC Peer-Out-Container. 21, 24–26, 32, 34, 51, 54, 73

RAM Random Access Memory. 18, 21, 45, 55, 58

RFID Radio-Frequency Identification. 15

RISC Reduced Instruction Set Computer. 45

ROM Read Only Memory. 18, 21, 45, 84

RSSI Received Signal Strength Indication. 83

RTC Real-Time Clock. 10, 37, 38, 46, 51, 55, 57

SPI Serial Peripherial Interface. 56–58

TTL Time To Live. 23, 24, 49–51, 54

TTS Time To Start. 22–25, 49, 50, 54, 73–75, 77

UART Universal Asynchronous Receiver Transmitter. 56, 57

WSAN Wireless Sensor and Actor Network. 1, 2, 7–10, 13, 14, 16, 18–21, 29, 35, 37, 38, 69, 80, 84, 85, 88, 91

WSN Wireless Sensor Network. 1, 7, 8, 14–16, 20, 41, 88

Bibliography

[1] I.F. Akyildiz and I.H. Kasimoglu. “Wireless sensor and actor networks: research challenges”. In: *Ad Hoc Networks* 2.4 (2004), pp. 351–367.

[2] ZigBee Alliance. *ZigBee PRO specification*. Tech. rep. Alliance, ZigBee, 2007.

[3] E. Baccelli et al. “RIOT OS: Towards an OS for the Internet of Things”. In: *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*. IEEE. 2013, pp. 79–80.

[4] J. Barbarán et al. “Tc-wsans: A tuple channel based coordination model for wireless sensor and actor networks”. In: *Computers and Communications, 2007. ISCC 2007. 12th IEEE Symposium on*. IEEE. 2007, pp. 173–178.

[5] M. Bordignon et al. “Seamless Integration of Robots and Tiny Embedded Devices in a PEIS-Ecology”. In: *Proc of the International Conference on Intelligent Robots and Systems (IROS)*. 2007.

[6] M. Broxvall, B.-S. Seo, and W. Kwon. “The PEIS kernel: a middleware for ubiquitous robotics”. In: *Proc. of the IROS-07 Workshop on Ubiquitous Robotic Space Design and Applications*. 2007.

[7] M. Broxvall et al. “PEIS Ecology: Integrating Robots into Smart Environments”. In: *Proc. of the 2006 IEEE Int’l Conf. on Robotics and Automation*. 2006, pp. 212–218.

[8] M. Ceriotti et al. “Is There Light at the Ends of the Tunnel? Wireless Sensor Networks for Adaptive Lighting in Road Tunnels”. In: *10th Int’l Conf. on Information Processing in Sensor Networks (IPSN)*. 2011.

[9] M. Ceriotti et al. “Monitoring Heritage Buildings with Wireless Sensor Networks: The Torre Aquila Deployment”. In: *8th Int’l Conf. on Information Processing in Sensor Networks (IPSN)*. Apr. 2009, pp. 277–288.

[10] P. Costa et al. “TeenyLIME : Transiently Shared Tuple Space Middleware for Wireless Sensor Networks”. In: *Proceedings of the international workshop on Middleware for sensor networks*. ACM, 2006, pp. 43–48.

[11] S. Craß, e. Kühn, and G. Salzer. “Algebraic Foundation of a Data Model for an Extensible Space-Based Collaboration Protocol”. In: *Int. Database Engineering and Applications Symposium (IDEAS)*. ACM, 2009.

[12] S. Craß et al. “Securing a Space-Based Service Architecture with Coordination-Driven Access Control”. In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)* 4.1 (2013), pp. 76–97.

[13] M. Csuk. “Developing an interactive, visual monitoring software for the peer model approach”. Master’s Thesis. Vienna University of Technology, Institute of Computer Languages, 2014.

[14] M. Díaz, B. Rubio, and J.M. Troya. “A Coordination Middleware for Wireless Sensor Networks”. In: *Proc. of the 2005 Systems Communications (ICW)*. 2005.

[15] M. Díaz, B. Rubio, and J.M. Troya. “A tuple channel-based coordination model for parallel and distributed programming”. In: *Journal of Parallel and Distributed Computing* 67.10 (Oct. 2007), pp. 1092–1107.

[16] M. Díaz, B. Rubio, and J.M. Troya. “TCMote : A Tuple Channel Coordination Model for Wireless Sensor Networks”. In: *Int’l Conf. on Pervasive Services (ICPS)*. 2005, pp. 1–4.

[17] L.-F. Ducreux et al. “Resource-based middleware in the context of heterogeneous building automation systems”. In: *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*. IEEE, Oct. 2012, pp. 4847–4852.

[18] A. Dunkels, B. Gronvall, and T. Voigt. “Contiki - a lightweight and flexible operating system for tiny networked sensors”. In: *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE, Nov. 2004, pp. 455–462.

[19] ETSI. *Final draft ETSI EN 300 220-1 V2.4.1 (2012-01)*. Tech. rep. ETSI, 2012.

[20] P. T. Eugster et al. “The many faces of publish/subscribe”. In: *ACM Computing Surveys (CSUR)* 35.2 (2003), pp. 114–131.

[22] M. Fassl. “Low-Power Radio Communication with Architecture-Independent Microcontroller Applications”. Bachelor’s Thesis. Vienna University of Technology, Institute of Computer Languages, 2014.

[23] T. Fjellheim. “A Coordination-Based Framework for Reconfigurable Mobile Applications”. PhD thesis. University of Tromso, 2006.

[24] T. Fjellheim. “Over-the-air deployment of applications in multi-platform environments”. In: *Software Engineering Conference, 2006. Australian*. IEEE, Apr. 2006, 10–pp.

[25] T. Fjellheim, S. Milliner, and M. Dumas. “Middleware support for mobile applications”. In: *International Journal of Pervasive Computing and Communications* 1.2 (2005), pp. 75–88.

[26] T. Fjellheim et al. “A process-based methodology for designing event-based mobile composite applications”. In: *Data & Knowledge Engineering* 61.1 (Apr. 2007), pp. 6–22.

[27] T. Fjellheim et al. “The 3DMA Middleware for Mobile Applications”. In: *EUC 2004, LNCS 3207*. 2004, pp. 312–323.

[28] C.-L. Fok. “Adaptive Middleware for Resource-Constrained Mobile Ad Hoc and Wireless Sensor Networks”. PhD thesis. Washington University in St. Louis, 2009.

[29] C.-L. Fok, G. Roman, and C. Lu. “Mobile agent middleware for sensor networks: an application case study”. In: *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*. IEEE, Apr. 2005, pp. 382–387.

[30] C.-L. Fok, G. Roman, and C. Lu. “Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications”. In: *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*. IEEE, June 2005, pp. 653–662.

[31] C.-L. Fok, G.-C. Roman, and C. Lu. “Agilla: A Mobile Agent Middleware for Self-Adaptive Wireless Sensor Networks”. In: *ACM Transactions on Autonomous and Adaptive Systems* 4.3 (July 2009), pp. 1–26.

[32] B. Ford. “Packrat parsing: a practical linear-time algorithm with backtracking”. PhD thesis. Massachusetts Institute of Technology, 2002.

[33] B. Ford. “Parsing Expression Grammars: A Recognition-based Syntactic Foundation”. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’04. Venice, Italy: ACM, 2004, pp. 111–122.

[34] D. Gelernter. “Generative Communication in Linda”. In: *ACM Trans. Program. Lang. Syst.* 7.1 (Jan. 1985), pp. 80–112.

[35] Carlo Ghezzi, Gianpaolo Cugola, and Gian Pietro Picco. “PeerWare: A Peer-to-Peer Middleware for Mobile TeamWork”. In: *ERCIM News* 54 (2003).

[36] S. Hadim, J. Al-Jaroodi, and N. Mohamed. “Trends in middleware for mobile ad hoc networks”. In: *Journal of Communications* 1.4 (2006), pp. 11–21.

[37] K. Herrmann. “MESHMd1- a middleware for self-organization in ad hoc networks”. In: *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*. IEEE, May 2003, pp. 446–451.

[38] G. Holasek. “Evaluation of the Peer Model Framework on a RCM4300 Evaluation Board”. Bachelor’s Thesis. Vienna University of Technology, Institute of Computer Languages, 2014.

[39] ISO. *Information technology – Syntactic metalanguage – Extended BNF*. Standard. International Organization for Standardization, 1996.

[40] ISO. *Information technology – Programming languages – Ada*. Standard. International Organization for Standardization, 2012.

[41] ITU. *V.250: Serial asynchronous automatic dialling and control*. International Telecommunications Union. July 2003.

[42] e. Kühn, S. Craß, and T. Hamböck. “Approaching Coordination in Distributed Embedded Applications with the Peer Model DSL”. In: *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*. Aug. 2014, pp. 64–68.

[43] e. Kühn, S. Craß, and G. Schermann. “Extending a Peer-based Coordination Model with Composable Design Patterns”. In: *Parallel, Distributed, and Network-Based Processing (PDP), 2015 23rd EUROMICRO Conference on*. Mar. 2015.

[44] e. Kühn, R. Mordinyi, and C. Schreiber. “An extensible space-based coordination approach for modeling complex patterns in large systems”. In: *Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2008, pp. 634–648.

[45] e. Kühn et al. “Flexible Modeling of Policy-Driven Upstream Notification Strategies”. In: *29th Symposium On Applied Computing (SAC)*. ACM, 2014.

[46] e. Kühn et al. “Peer-Based Programming Model for Coordination Patterns”. In: *15th Int. Conf. on Coordination Models and Languages (COORDINATION)*. LNCS. Springer, 2013, pp. 121–135.

[47] P. Levis et al. “TinyOS: An Operating System for Sensor Networks”. In: *Ambient intelligence*. Springer, 2005, pp. 115–148.

[49] M. Mamei, R. Quaglieri, and F. Zambonelli. “Making Tuple Spaces Physical with RFID Tags”. In: *Proceedings of the 2006 ACM Symposium on Applied Computing*. 2006, pp. 434–439.

[50] A. Marek. “Design and implementation of TinySpaces: the .NET micro framework based implementation of XVSM for embedded systems”. Master’s Thesis. Vienna University of Technology, Institute of Computer Languages, 2010.

[53] M. Mernik, J. Heering, and A.M. Sloane. “When and How to Develop Domain-specific Languages”. In: *ACM Comput. Surv.* 37.4 (Dec. 2005), pp. 316–344.

[54] R. Mordinyi, Evae. Kühn, and A. Schatten. “Space-Based Architectures as Abstraction Layer for Distributed Business Applications”. In: *2010 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*. IEEE Computer Society, Feb. 2010, pp. 47–53.

[55] A.L. Murphy and G.P. Picco. “Transiently Shared Tuple Spaces for Sensor Networks”. In: *Proc. of the Euro-American Workshop on Middleware for Sensor Networks*. 2006.

[56] F. Pacull et al. “Self-organisation for Building Automation Systems: Middleware LINC as an Integration Tool”. In: *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, Nov. 2013, pp. 7726–7732.

[57] G.-P. Picco, A.L. Murphy, and G.-C. Roman. “LIME: Linda meets mobility”. In: *Int. Conf. on Software Engineering (ICSE)*. ACM, 1999.

[58] D. Rauch. “PeerSpace.NET: implementing and evaluating the Peer Model with focus on API usability”. Master’s Thesis. Vienna University of Technology, Institute of Computer Languages, 2014.

[59] A. Saffiotti and M. Broxvall. “PEIS Ecologies : Ambient Intelligence meets Autonomous Robotics”. In: *Proc. of the sOc-EUSAI (Smart Objects and Ambient Intelligence) conf.* 2005.

[60] A. Saffiotti et al. “The PEIS-Ecology Project: Vision and Results”. In: *IEEE/RSJ Int’l. Conf. on Intelligent Robots and Systems (IROS)*. 2008, pp. 22–26.

[61] G. Schermann. “Extending the Peer Model with composable design patterns”. Master’s Thesis. Vienna University of Technology, Institute of Computer Languages, 2014.

[63] G. Van Rossum and F.L. Drake. *The python language reference manual*. Network Theory Ltd., 2011.

[64] G. Vigna. “Mobile agents: ten reasons for failure”. In: *Mobile Data Management, 2004. Proceedings. 2004 IEEE International Conference on*. IEEE, 2004, pp. 298–299.

[65] R.P. Weicker. “Dhrystone: A Synthetic Systems Programming Benchmark”. In: *Commun. ACM* 27.10 (Oct. 1984), pp. 1013–1030.

[66] D. A. Wheeler. “Ada, C, C++, and Java vs. The Steelman”. In: *Ada Lett.* XVII.4 (July 1997), pp. 88–112.

[67] J. Yick, B. Mukherjee, and D. Ghosal. “Wireless sensor network survey”. In: *Computer Networks* 52.12 (2008), pp. 2292–2330.

Patents

[21] E.F. Færevaag. “Advanced Energy Profiler”. Patent US20110087908 A1. Apr. 2011. URL: https://www.lens.org/lens/patent/US_2011_0087908_A1.

Webpages

[48] Libelium. *Waspmote Datasheet*. www.libelium.com/development/waspmote/documentation/waspmote-datasheet/?action=download. Last Accessed: 1-Apr-2015.

[51] Memsic. *MICAz - Memsic*. http://www.memsic.com/userfiles/files/Datasheets/WSN/micaz_datasheet-t.pdf. Last Accessed: 1-Apr-2015.

[52] Memsic. *TELOSB - Memsic*. http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf. Last Accessed: 1-Apr-2015.

[62] Seeedstudio. *Seeeduino Stalker v2.3 - Wiki:* http://www.seeedstudio.com/wiki/Seeeduino_Stalker_v2.3. Last Accessed: 1-Apr-2015.